TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SANTERI SUVANTO

VISUALIZING A CONTINUOUS DELIVERY PIPELINE

Master of Science Thesis

Examiner: Prof. Timo Hämäläinen

Subject and examiner approved by Faculty of Computing and Electrical Engineering Council meeting on 30th of November 2017

ABSTRACT

FACULTY OF COMPUTING AND ELECTRICAL ENGINEERING:
Visualizing a Continuous Delivery Pipeline

Tampere University of Technology
Master of Science Thesis, 45 pages, 7 Appendix pages
May 2018
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Timo Hämäläinen

Keywords: Visualization, Continuous Delivery, Grafana

The purpose of this work is to provide a visualization service for the verification process of a continuous software delivery pipeline. The visualization service is required to gather data from various other services inside the verification process and visualize it as a time series over extensive periods of time. It is expected to provide simple and illustrative views to the overall status of the pipeline and the software under verification. The contents of this paper are targeted at anyone interested in utilizing the provided service.

The visualization service was implemented as a system named Data Visualizer. Collection of the data was implemented by utilizing a generic XML based format along with a separate metadata description, which enabled a modular framework with highly simplified system configuration. InfluxDB and Grafana were used as the storage and visualization components inside this framework. The system was packaged and productized using the Docker environment.

The resulting Data Visualizer system fulfilled the work requirements and was integrated with the pipeline environment without notable problems. Furthermore, the implementation was generic enough to be used to visualize any time series data from any environment, provided that a data format description and a data parsing component was implemented for the given source of the data.

Currently, the Data Visualizer system is only used in a reference implementation of a software verification pipeline. The reference pipeline itself is still under development, meaning the user base for Data Visualizer will remain small until the entire pipeline is productized and distributed across the organization. Meanwhile, alternative use cases for the system will be sought to gather user feedback for further development.

TIIVISTELMÄ

TIETO- JA SÄHKOTEKNIIKAN TIEDEKUNTA:
Ohjelmiston Jatkuvan Toimituksen Visualisointi

Tampereen teknillinen yliopisto
Diplomityö, 45 sivua, 7 liitesivua
Toukokuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastaja: Prof. Timo Hämäläinen

Avainsanat: Visualisointi, Jatkuva Toimitus, Grafana

Työn tarkoituksen oli tuottaa visualisointipalvelu ohjelmiston jatkuvan toimituksen prosessiin, ohjelmiston varmennuksen yhteyteen. Palvelun tehtävänä on kerätä dataa varmennusprosessin monista eri palveluista ja visualisoida sitä aikasarjana pitkillä aikaväleillä. Visualisoinneilta vaadittiin selkeyttä ja havainnollistavaa näkökulmaa varmennusprosessin ja sen käsittelemän ohjelmiston tilan arviointiin. Tekstin sisältö on tarkoitettu kaikille palvelun käytöstä kiinnostuneille.

Visualisointipalvelu toteutettiin järjestelmänä, joka sai nimen Data Visualizer. Datan keräys rakennettiin yksinkertaisen XML formaatin ympärille, jota täydennettiin erillisellä metadatan kuvauksella. Tämä lähestymistapa mahdollisti järjestelmän rakentamisen modulaarisen rungon ympärille, mikä edelleen yksinkertaisti järjestelmän konfiguraatioiden hallintaa. Järjestelmän alustaviksi varastointi- ja visualisointikomponenteiksi valittiin InfluxDB ja Grafana. Järjestelmä paketoitiin ja tuotteistettiin hyödyntäen Docker ympäristön tarjoamia palveluja.

Toteutettu järjestelmä täytti annetut vaatimukset ja integroitiin osaksi varmennusprosessia, eikä järjestelmän käyttöönotossa havaittu huomattavia ongelmia. Toteutus mahdollisti aikasarjadatan keruun myös mistä tahansa muusta lähteestä. Vaatimuksena uusien lähteiden lisäämiselle oli ainoastaan lähteen metadatan kuvauksen rakentaminen sekä datanäytteiden jäsentelijän kehittäminen.

Data Visualizer järjestelmä on toistaiseksi käytössä vain työn teettäneen organisaation mallitoteutuksessa ohjelmiston varmennusprosessille. Tämä mallitoteutus on itsessäänkin vielä kehitysvaiheessa, joten Data Visualizer järjestelmän käyttäjäkunta tulee pysymään pienenä, kunnes malliprosessi on tuotteistettu ja toimitettu organisaation eri ryhmille. Kehitystyön ohessa Data Visualizer järjestelmälle etsitään myös vaihtoehtoisia käyttökohteita, jotta kattavan käyttäjäpalautteen kerääminen olisi mahdollista.

PREFACE

This work started as a continuation to a summer trainee position that suddenly extended to over a year of work around the same continuous delivery process I dealt with during the work itself. It served as a good learning process to a massive number of different technologies and real world working practices.

The team I worked with was extremely supportive during both the coding and writing phases of the work and shared a lot of their experience and insight for me to learn from. Working with them was inspiring and I would like to thank them for that.

21.5.2018, Tampere

Santeri Suvanto

TABLE OF CONTENTS

ABBREVIATIONS

API	Application Programming Interface
DB	DataBase
CD	Continuous Delivery
CI	Continuous Integration
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
REST	REpresentational State Transfer
XML	eXtensible Markup Language
XSD	XML Schema Description
YAML	YAML Ain't Markup Language

1. INTRODUCTION

Continuous delivery systems have successfully increased the productivity, reliability and deployment speed of large and complex software projects through adoption of efficient and highly automated verification processes and fast feedback cycles. However, the overall status of these processes can easily become obscured, making the detection of potential risks more difficult.

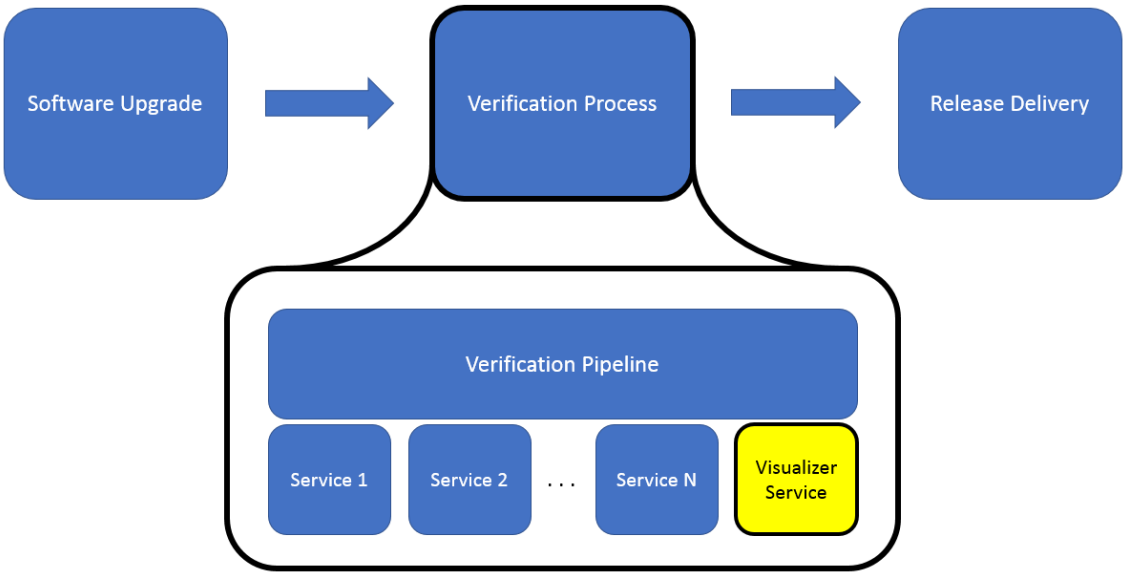

Figure 1.1 Visualizer service in big picture

The purpose of this work is to provide a new visualizer service component for the software verification pipeline presented in figure 1.1. This component will gather data from various other services of the verification pipeline and visualize it over extensive periods of time. It should provide simple and illustrative views to the overall status of the pipeline and the software under verification.

The structure of this thesis is as follows. Chapter 2 gives a brief introduction to topics related to Continuous Delivery, DevOps and data analysis. Chapter 3 continues with the description of the monitoring service and the design choices made during the development. Chapter 4 then analyzes the implementation results and gives example use cases for the service. Finally, Chapter 5 concludes the topic of this work and presents possibilities for further development.

2. CONTINUOUS DELIVERY AND DEVOPS

It is important to understand the principles of continuous delivery before starting to work on the visualization of such systems. This chapter will briefly describe the key aspects of continuous delivery to give the reader basic understanding of the presented problem. To explain the benefits of monitoring and visualizing a continuous delivery system, the principles of the DevOps practices are also explained. Finally, the issue of data visualization is discussed in a wider scope.

2.1 Continuous Delivery

Speed, agility, and reliability are crucial factors when delivering software to the customer. Historically, large scale software projects have had substantially high risks that often cause a significant rise of costs from the initial project plan. Continuous delivery aims to increase the productivity of software projects by decreasing the cycle time from writing software changes to building a deployment-ready release candidate [1,2].

Figure 2.1 Software delivery process

Continuous delivery is achieved by utilizing automation and agile development methods to create a continuous integration process that starts from committed changes and leads all the way to the end of the software delivery process. This increases the reliability and predictability of each release as it minimizes the effect of human factors. [1,2]

The automated process from development to deployment is defined as a deployment pipeline. Every software change starts a new pipeline process where the stages of figure 2.1 are executed in order. The process ends with either a detected fault in the software or a new qualified release candidate. [1,2]

The process of build and testing automation is also known as continuous integration [3]. To expand from continuous integration to continuous delivery, the delivery process must be automated as well. The following subchapters will briefly describe these methods.

2.1.1 Continuous Integration

Continuous integration is the basis of continuous deployment and has been in use for many software projects. It means that the written code should be committed and integrated to the main branch as often as possible to avoid difficulties related to large scale software integration [3]. The continuous aspect of this process is achieved through building and testing automation.

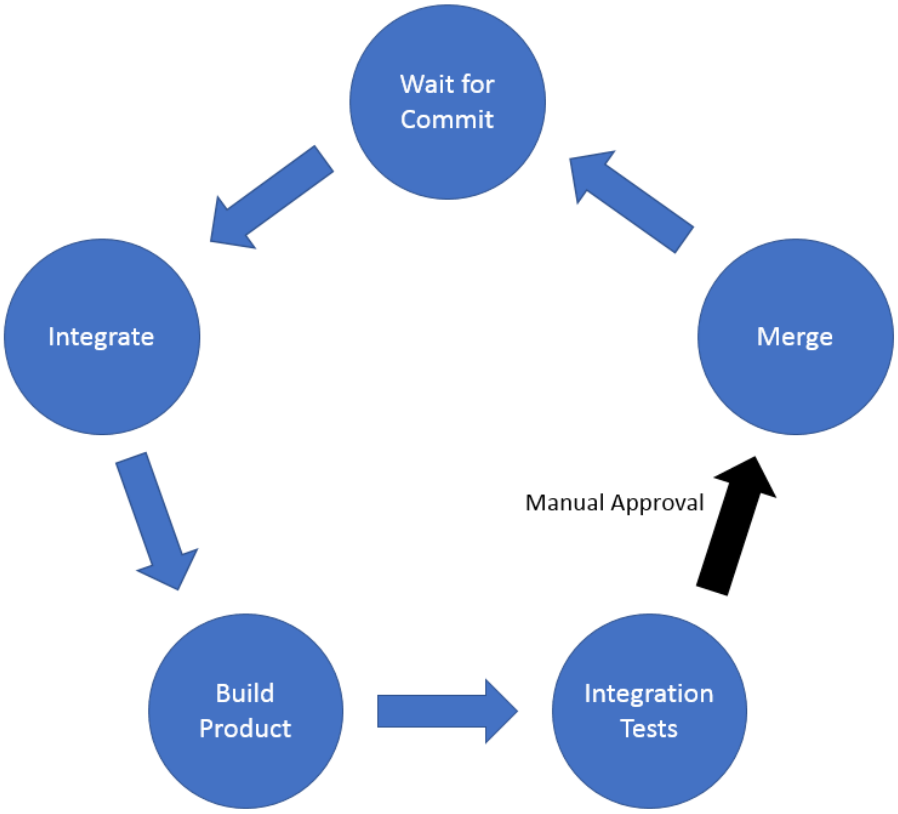

Figure 2.2 Continuous integration cycle

The continuous integration cycle is visualized in figure 2.2. It starts when a committed change reaches the integration server. The change is then integrated into a separate branch where it can be tested. The integration server may run for example unit tests before integrating the change to quickly detect basic environment dependent issues such as missing modules or incorrect build tool versions.

When the change is integrated to the product under development, the entire product must be built and tested. These phases ensure that the committed change works as intended from the perspectives of other product components. When the size of the change is kept small, the task of solving found issues becomes easier as well. Finally, the change can be merged into the main branch and prepared for the delivery process, usually through a manual approval process. [1]

Every committed change goes through an automated process that runs tests on both unit and system levels. Therefore, the entire software must be built in a production-like environment for every change. The use of this process proves that the system under development is constantly in a both buildable and working state. [1]

When as much of the testing is done as early as possible, the cost of fixing the discovered faults becomes more straightforward. Taking the system integration as part of the development process from the beginning keeps any issues that may occur manageable and decreases the feedback time between committing a change and detecting a fault in the integration process. [1]

2.1.2 Delivery Automation

In addition to build and test automation, continuous delivery aims to automate the delivery process as well. When the software successfully passes the continuous integration stage, it continues to further acceptance testing in a production-like environment. Builds that pass these tests can then be archived and finally released. [1]

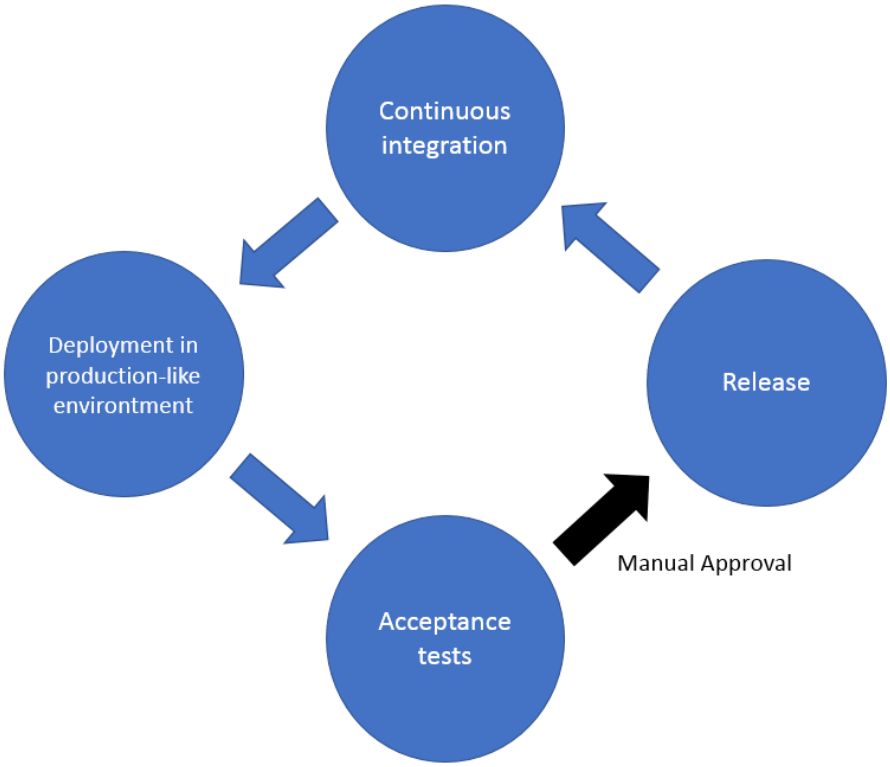

Figure 2.3 Continuous delivery cycle

Figure 2.3 presents the continuous delivery cycle as an extension to the continuous integration process described in figure 2.2. After a successful integration cycle, the product must be tested in a production-like environment before release.

The purpose of these phases is to simulate the user environments and run extensive testing on both functional and non-functional properties of the product to provide reliable proof that the committed change has not broken anything. Changes that pass this phase are therefore ready to be released and deployed to the end users. [1]

Keeping track of all the production environments as well as the different system configurations available must be handled by the delivery process. Various configuration management tools may be used for this purpose with various levels of automation depending on the complexity and number of the system configurations. [1]

2.2 DevOps

DevOps is a recently introduced term that describes the organizational changes required to lower certain risks involved with traditional software development methods. It extends from the agile development principles and is closely related to continuous delivery. DevOps methods can be used to achieve and improve continuous delivery while implementing continuous delivery can be a part of an organizational shift to DevOps. [4–7]

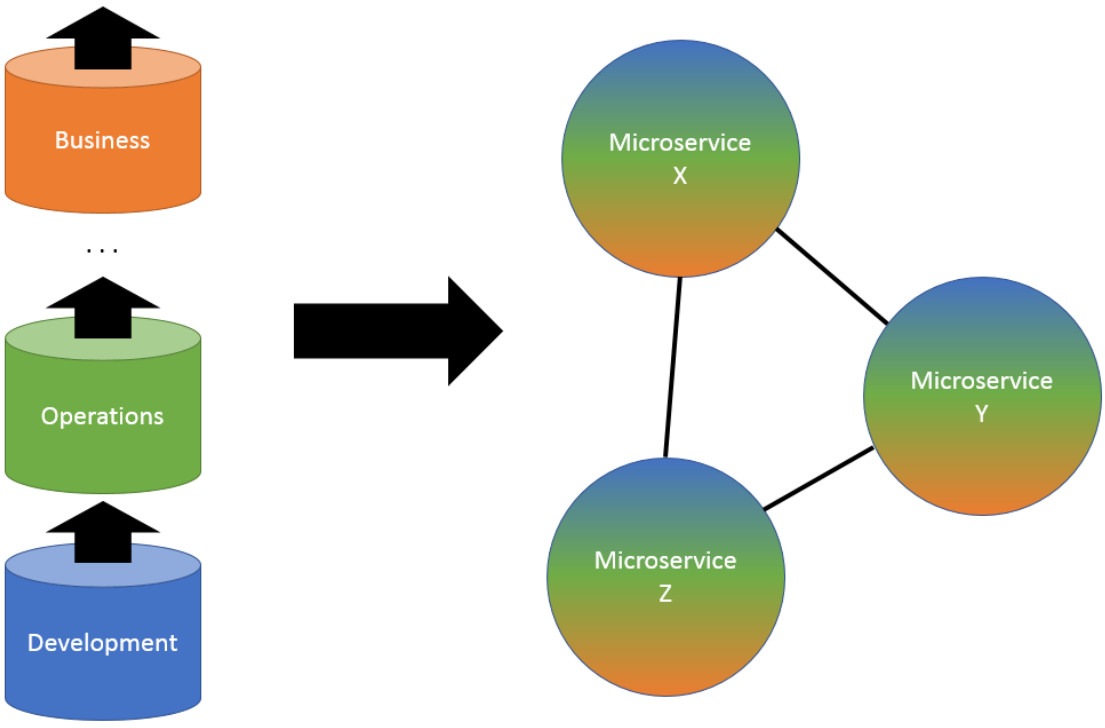

Figure 2.4 The organizational shift to DevOps

Numerous and sometimes even contradictory methodologies have been presented under the DevOps term, proven by various interviews and literature surveys on the subject [8,9]. A common idea related to the term is that the different siloed groups, such as development, operations, and business, should be integrated to form agile and cross-functional working environments as presented in figure 2.4.

This collaboration between the groups is said to improve the software development workflow and decrease the cycle time of the software delivery process. [5,7]

The shift from traditional software production methods to DevOps is often connected with the architectural design where the product under development is built on multiple components or microservices. Figure 2.4 shows how each microservice can be developed by a team consisting of people with various skillsets. As a result, each member of the team will learn at least some basics from the work and responsibilities of the other members, which helps to improve communication within the organization. [4]

2.2.1 Common Practices

Although DevOps can often be understood completely differently depending on the person who attempts to explain it, some common approaches to achieve the goal presented in figure 2.4 can be found. This chapter focuses on three of these approaches: speeding up the workflow from development to operations, creation of feedback loops from operations back to development, and building a high-trust culture within the organization to enable to improve sharing of knowledge.

The most widespread DevOps practice is to increase the speed of the development workflow by decreasing the cycle time from development to operations. Continuous integration and delivery are often used to achieve this behavior and it can be taken even further with continuous deployment. By keeping the work in progress at a manageable size, development teams can efficiently produce tested software at a faster pace. [4]

Another commonly used DevOps term is continuous feedback. By creating fast feedback loops from operations back to development, issues can be found earlier and they can be addressed immediately. This practice will prevent the problems from spreading further in the development workflow. [4]

Continuous monitoring and changes towards a high-trust culture help to transform locally found solutions into global changes for the entire organization. Instead of focusing on finding the person who caused a problem, it is more important to understand why the problem happened and how could it be prevented in future. When the people within the organization are more willing to share collectable information of their work, finding efficient solutions to common problems becomes easier. [4]

Especially the final approach can greatly benefit from added visualizations that raise awareness on critical issues. Sharing information also becomes more appealing when the effort required to create various visualizations is low. Enforcing a high-trust culture where data can be openly shared is not a direct goal of this work, although the implemented visualization service can be a useful tool in those situations.

2.2.2 DevOps and Monitoring

Monitoring is a crucial part of DevOps principles as it helps to discover and improve the feedback loops in the software delivery process. By combining data from every group within the organization, the overall status of the project can be efficiently monitored. However, obtaining this data may be difficult if the groups are unwilling to share it. [6]

Operations teams have traditionally done extensive monitoring on the state and stability of the application under development and the environment it runs on. This often requires the development team to implement various self-monitoring features to the developed product, which can lead to problems when communication between these two teams is inefficient. DevOps practices help to form a working environment where the collection of this data is as simple as possible. [6]

2.3 Data Collection and Visualization

Continuous delivery pipelines contain numerous possibilities for collection and visualization data. Anything from the results of automated test cases to the resource usage of each pipeline stage may contain helpful information for the various groups within the organization.

Different visualizable aspects of software can be divided into three categories: structure, behavior and evolution. Structural visualizations describe static information, such as code and data structures. Behavioral visualizations measure the execution of the software under various conditions while evolution refers to visualization of the development process. The focus of this work is on visualizations of behavioral aspects. [10]

Book by Aigner et al. introduces three different metrics that should be considered when creating a visualization process: expressiveness, effectiveness, and appropriateness. Expressiveness means that the visualizations should only focus on the required data. Effectiveness describes how easy it is for human viewers to process the visualizations while appropriateness measures the cost-benefit ratio of the visualization process itself. [11]

It is important to carefully decide both what data should be collected and how it should be visualized. Collected metrics should efficiently represent the status of the data source without unnecessary redundancy. Visualizations should also be clear and simple without being misleading or hiding crucial information. The importance of aesthetics is also emphasized in visualization systems. [12,13]

Visualization systems are quickly developing towards a more interactive experience. Being able to control the position of graph axes or give different parameters to affect the visualization result will increase both the expressiveness and effectiveness of created visualizations. Tools for data extraction and modification will also decrease the cost of creating new visualizations that may reveal previously hidden details. [14,15]

Even if the collected data might not be classified as Big Data, utilizing big data collection and analysis techniques will give some insight to the implementation of the visualization system. The proceeding subchapters will explore the possibilities in big data analysis through a reference model presentation.

2.3.1 Components

Numerous components may be involved in a data visualization system. In addition to gathering, storing, and visualizing data, various processing, analysis and management techniques can be applied. There also might be numerous other applications beyond the scope of data visualization where the gathered data can be taken into use.

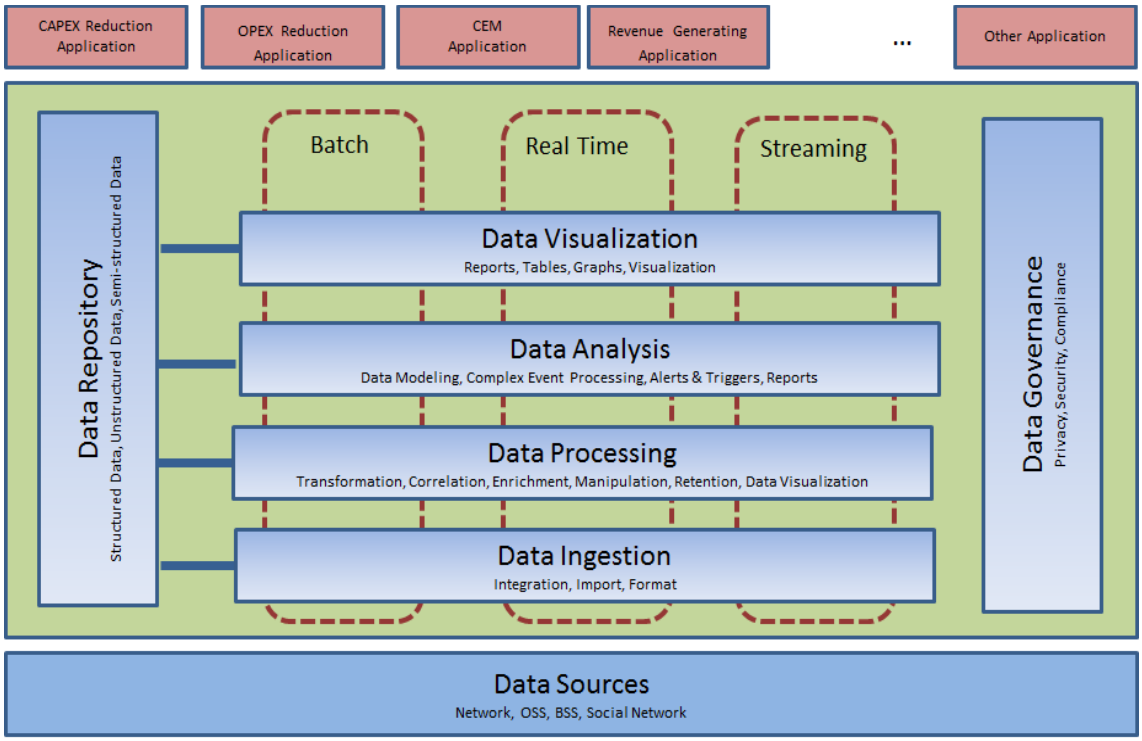

Figure 2.5 TM Forum Big Data analytics reference model [16]

Figure 2.5 presents a reference model for a full Big Data analytics system where the different components that might be implemented in such systems are displayed on separate layers [16]. In this model the components that are required for the task of this work are data ingestion, data repository and data visualization. Other components are out of the scope of this work, but may be implemented if needed.

2.3.2 Data Flow

The data might travel not only through multiple components but in different paths as well. The number of available paths for the data varies based on the number of implemented components. The type of the data flow may also vary depending on how it can be fed to the system.

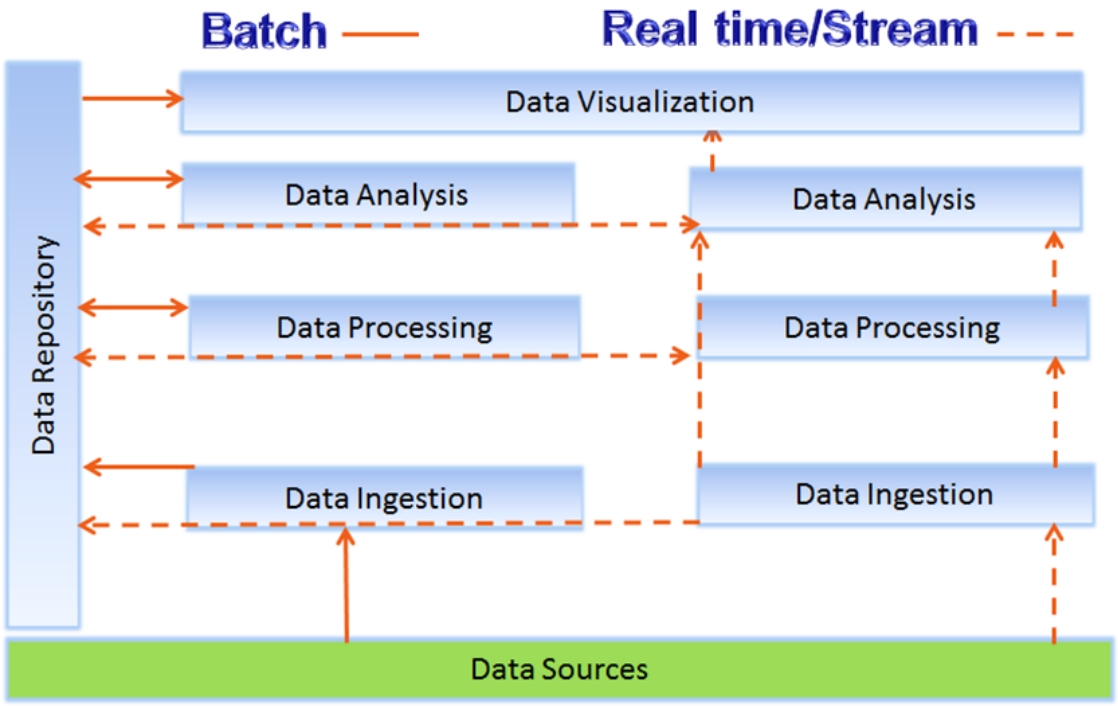

Figure 2.6 Data traffic in the TM Forum reference model [16]

The flow of data in the reference model is separated into batch, stream, and real-time data. The different routes these data types can take are visualized in figure 2.6. In this model the route that is required to be implemented starts from data sources, goes to data repository through data ingestion, and ends up in data visualization. In the scope of this work, the data flow is also limited to the batch-type. The other components, routes, and flow types may be useful for further development of the system. [16]

2.4 Related Work

Process visualization is a crucial subject on many fields of work. At least, it provides the basic means for sharing information about any process or pipeline. When taken further, it can be used for extensive data analysis and process modelling. All these methods help to optimize the process, either by raising awareness on critical issues that need to be solved or by presenting optimizations to existing implementations. This section presents works done on process visualization that describe the challenges and possibilities in more detail.

Survey conducted by Kandel et al. shows that the data analysts of organizations in the field of information technology have varying skillsets and means of performing data analysis and visualization. The paper introduces archetypes for data analysists and models for analysis workflows. Current trends in the industry and their effects to data analysis and visualization are also discussed. The paper states that by increasing the availability and quality of collectable data, quality and performance of the data analysis process can be improved. [17]

Visualization workflows are discussed in more theoretical detail in the article by Chen and Golan. The paper presents separate levels for visualization tasks depending on their complexity. Classes for visualization and data analysis workflows with emphasis on the separation of human and machine centric processes along with information-theoretic cost-benefit measurement methods. [18]

As presented in chapter 2.3, software projects have multiple aspects that can be visualized. Work by Caserta and Zendra presents visualization methods for the structural aspects of software. These aspects are not within the scope of this work but they are an equally important part of software visualization. [19]

The use of three-dimensional visualization techniques has become increasingly relevant as the capabilities and performance of visualization tools has increased. In their work, Teyseyre and Campo discuss the state of 3D software visualization in 2009. At the time many of the introduced visualization techniques were concluded to be on a prototype level and in need of further research. [20]

Measuring and visualizing progress and stability of coding work has been done in most software projects. Work by Staron et al. introduces some visualization methods for measuring stability of code changes. It gives some insight to how heatmaps may be used to present time oriented data in a way that makes detection of possibly important anomalies easier. [21]

Work by Arttu Leppäkoski and Timo D. Hämäläinen introduces a process mining tool with emphasis on visualization. The tool collects and stores data from various sources and visualizes it through a graphical web interface. The workflow and core architecture are highly similar to the application presented in chapter 3 of this paper, which reinforces the observation that there is strong demand for process visualization applications in the field of software development. [22]

Numerous commercial applications exist for various visualizations tasks. For example, applications by Seerene Inc. and Power BI tools by Microsoft offer visualizations for business analytics and project management [23,24]. The Elastic Stack is a collection of open source tools performing data collection, analytics, and visualization for a more generic use case [25].

3. DATA VISUALIZER DESIGN

The visualizer service for the software verification pipeline presented in figure 1.1 was implemented as a system named Data Visualizer. The purpose of this system was to provide tools for gathering data from various parts of the verification pipeline and its components and visualize it over extensive periods of time. Collection of the data was implemented by utilizing a generic XML based format in a batch-type data flow along with a separate data format description for system configuration.

Figure 3.1 Basic data flow of Data Visualizer

The basic flow of data in Data Visualizer is described in figure 3.1. It starts with a data formatting phase where the measured data is extracted from the verification pipeline and reformatted to the generic XML format. The formatted data is then sent through an HTTP API to the storage phase where it is converted and stored to a data container. Finally, the visualization phase pulls the data from the container and allows the user to view it through a user interface. These phases map directly to the data ingestion, data repository, and data visualization components of the TM Forum data analytics model described in chapter 2.3.1 [16].

A more detailed representation of this data flow is shown in figure 3.2. It shows the flow of the one-time configuration data as white arrows and the following runtime data as blue arrows. The configuration data is based on the data format description that is designed by exploring and analyzing a source of data in the verification pipeline. When the data format description is complete, a data parser can be developed to convert the data into the generic XML format.

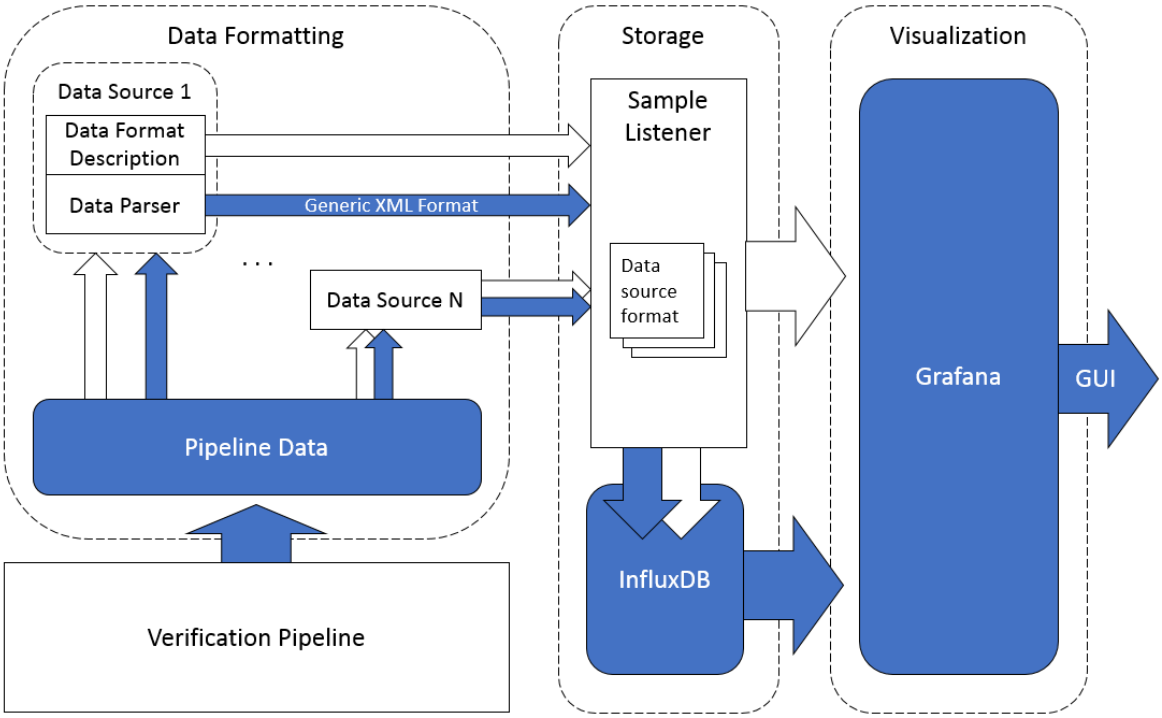

Figure 3.2 Detailed data flow of Data Visualizer

The following subchapters describe the various aspects of Data Visualizer in more detail. The three phases presented in figures 3.1 and 3.2 are explained in more detail in chapters 3.1, 3.2, and 3.3. Chapter 3.4 describes the system setup process and the possibilities for its automation. Extensions made to the system are discussed in chapter 3.5.

3.1 Data Formatting

The Data Visualizer system provides various data parsers that gather time series data from a data source in the verification pipeline and reformat it to a generic XML format. The purpose of this format is to provide a uniform interface between the data formatting and storage phases. It removes any dependencies to the implementation of the storage phase and allows the data parsers of various data sources to be developed separately from the rest of the system.

All storage specific information is separated to a data source format description that is used to automatically configure the system for the data parsers. The description is written in XML due to consistency with the sample data. Python *lxml* library provides simple and efficient methods for parsing and validating these formats. XML also allows a visually clear representation of sample data as well as efficiency and robustness for API communication. Furthermore, use of an XML Schema Description provides a standard validation method for the data format description.

Since the visualization requirement of the system is to display data values against a time axis, the parsed data always contains a time series. This affects the data formatting as each sample must contain a timestamp with a consistent format and a value that matches with the rest of the data.

3.1.1 Sample format

A parsed sample XML contains a root element with the name of the data source as its tag. Under the root element is a list of sample elements, each having the name of the sample as the element tag and measured values as the element attributes. Each sample must have at least an integer timestamp and one other measured value. An example sample for a data source named *example_pipeline* with sample elements named *build* and *phase* is presented in figure 3.3.

```
<example_pipeline>
     <build name=”example_build1” duration=”10” timestamp=”1” >
          <phase name=”phase1” duration=”1” timestamp=”2” />
          <phase name=”phase2” duration=”3” timestamp=”4” />
          <phase name=”phase3” duration=”5” timestamp=”5” />
     </build>
     <build name=”example_build2” duration=”99” timestamp=”1” >
          <phase name=”phase1” duration=”10” timestamp=”5” />
          <phase name=”phase2” duration=”80” timestamp=”15” />
     </build>
</example_pipeline>
```

Figure 3.3 Example sample

As seen in figure 3.3, the sample elements may also contain subsamples listed inside them indicating a connection between subsamples and their parent sample. In the example sample each build element contains phase elements as subsamples. Chapter 3.1.2 will explain the functionalities of this sample structure in more detail.

3.1.2 Data format description

Before the data parser in figure 3.2 can send the formatted data to the sample listener, the storage phase needs to know how the data should be stored. This can be done by creating a data format description for the data source. The XML schema of the format description can be found in appendix A and an example format description for the sample data defined in chapter 3.1.1 is presented in figure 3.4.

```
<datasource name=”example_pipeline” timestamp_precision=”s”
            expire_days=”365”>
     <sample name=”build”>
          <identifiers>
               <attribute name=”name” type=”str” />
          </identifiers>
          <measurements>
               <attribute name=”duration”
                          type=”int” min=”0” />
          </measurements>
          <children reference_identifiers=”name”>
               <sample name=”phase”>
                    <identifiers>
                         <attribute name=”name” type=”str” />
                    </identifiers>
                    <measurements>
                         <attribute name=”duration”
                                    type=”int” min=”0” />
                    </measurements>
               </sample>
          </children>
     </sample>
</datasource>
```

Figure 3.4 Example format

The format XML has a root element with tag *datasource* and attributes *name* and *timestamp_precision* with the name of the data source and the precision of the integer timestamps as their values. An optional *expire_days* can also be added to configure automatic deletion of samples older than the given number of days. Under the format root is a list of sample format subtrees, each containing information about a sample from the data source.

Each sample format subtree consists of a root element that defines the name of the sample. The root contains three subcategories: identifiers, measurements, and children. The first two of these categories contain definitions for the sample element attributes. Each of these definitions tells the name and type of the attribute value as well as any other restrictions it might have. For example, the *duration* attribute of a sample named *build* is defined as an integer with a minimum value of 0 in the example format of figure 3.4.

The identifiers category contains definitions for sample attributes that are used to identify a specific type of sample in the series while the measurements category contains attribute definitions for the actual measured data. For example, the identifiers would be used in the WITH-section of an SQL type database query while the measurements would be used the SELECT-section.

The children category contains a list of sample definition trees for any subsamples the defined sample might have. Samples of same type may also be nested within each other if the children list contains an empty element with tag *self*.

Identifier attributes used to identify a child-parent relationship must be defined in an attribute named *reference_identifiers* in children category element. The value of this attribute is a comma separated list of values that should match the name value of the attribute definition in the identifiers category.

3.2 Storage

After configuration of the data source formats defined in chapter 3.1.2 the storage phase receives the XML formatted samples presented in chapter 3.1.1 and converts them to entries in a storage container. The data can then be queried from the container and viewed in the visualization phase.

This storage phase uses the data source formats to generate automatic configurations for the entire system. These configurations are managed by the sample listener component introduced in chapter 3.2.2.

3.2.1 Storage Container

As defined in chapter 3.1.1, the data received by the storage phase is a series of samples containing various measurements and timestamps. Therefore, a container that specializes in such time series data is the optimal solution for the storage phase. Storing the data in a more primitive container implementation would require extensive coding effort to create a suitable interface for the Data Visualizer framework.

After researching available storage systems, an open source time series database developed by InfluxData named InfluxDB was chosen to be used as the basis of the storage phase. Simple setup procedure, high read and write performance, and previous experience with the database were the main reasons for this choice. Prometheus and Elasticsearch were also considered, but as they focus on much more than just the storage of time series data, their implementations in Data Visualizer were postponed as possible continuations to this work. [26-28]

InfluxDB consists of storage instances called *databases* that contain measurements. Each measurement entry is separated to a name, tags, fields, and a timestamp. Tags are designed to contain indexed metadata that is used to identify the measurement while the actual measurement data is stored in fields. Tags are always interpreted as strings while fields are defined with a type. [29]

Data in InfluxDB is read and written through an HTTP API with an SQL-like query language named InfluxQL. Similarly to SQL, InfluxQL queries have SELECT, FROM, WHERE, and GROUP BY sections. The SELECT clause defines the tags and fields to be read while the FROM clause finds the measurement by name. The tags of a measurement have higher performance compared to fields when used in the WHERE and GROUP BY sections of the query. Writing data is done with the INSERT clause where the name, tags, fields, and the timestamp of a measurement are simply listed in the query. [29]

3.2.2 Sample Listener

The sample listener acts as the HTTP API between the data formatting and storage phases. It also automatically configures the storage container and the visualization phase with the sample definitions read from the format XMLs defined in chapter 3.1.2. The component is written in Python 3 and the overview of the system is displayed in figure 3.5. Similarly to figure 3.2, the flow of configuration data is shown in white and sample data in blue.

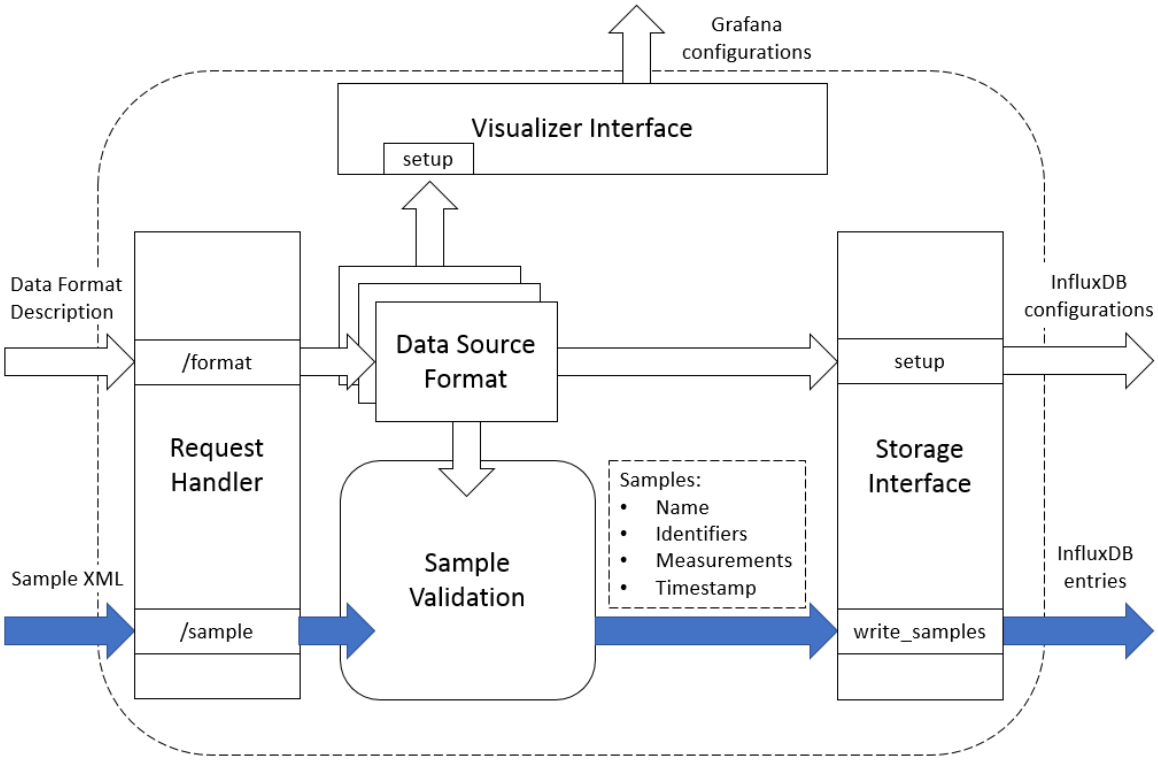

Figure 3.5 Sample listener

Before any samples can be written to the system, the format XML of the data source must be sent to the listener API using an HTTP POST request on the */format* endpoint. After the format has been validated and stored on the system, it will be used to configure InfluxDB and the visualization phase for writing sample data.

When an HTTP POST request is received on the */sample* endpoint the request payload containing the samples is validated using the corresponding sample format and converted to an intermediate structure. This structure consists of the sample name and lists of name-value pairs for the sample identifier- and measurement attributes as well as any reference identifiers passed by possible parent samples.

Both system configurations as well as the data samples are sent through a generic interface that implements the necessary operations for the underlying component. Configurations for both interfaces are performed using a method named *setup* while samples can be written using the *write_samples* method of the storage interface. The purpose of these interfaces is to separate the component specific functionalities from the rest of the system, making possible replacement of each component as simple as writing a new interface subclass.

3.3 Visualization

In the visualization phase, the content of the InfluxDB storage described in chapter 3.2.1 is visualized in a GUI. Since the storage phase uses InfluxDB through a generic interface, the visualization tool should be able to take advantage of the added modularity as well.

An open source visualization tool named Grafana was chosen for this task. Compatibility with numerous data sources, increased functionality offered by plugins, and previous experience with the tool were the main reasons for this choice. With built-in support for InfluxDB, it was simple to set up and configure in the Data Visualizer system. [30]

3.3.1 Dashboards

Visualization in Grafana is implemented using configurable dashboard pages. They define a GUI that is built using premade panel blocks that can be configured to view data from configured data sources. Dashboards can be created and modified by users with administrator or editor role and viewed by everyone.

Dashboard panels can be configured with alerts that change their state on predefined time intervals depending on the values of the data that is being viewed. These states can be connected to the notification system in Grafana, which allows custom messages to be sent to different endpoints, for example an e-mail address or a custom webhook endpoint.

3.3.2 Configuration

Before a data source can be used in dashboards, it needs to be configured in Grafana by an administrator. This can be done either manually with the GUI or through an HTTP API. Since the format XML files used in the storage phase describe the structure of the data in the storage container, they can be used to configure the Grafana data sources as presented in figures 3.5 and 3.6. This reduces configuration requirements from the user of the system significantly.

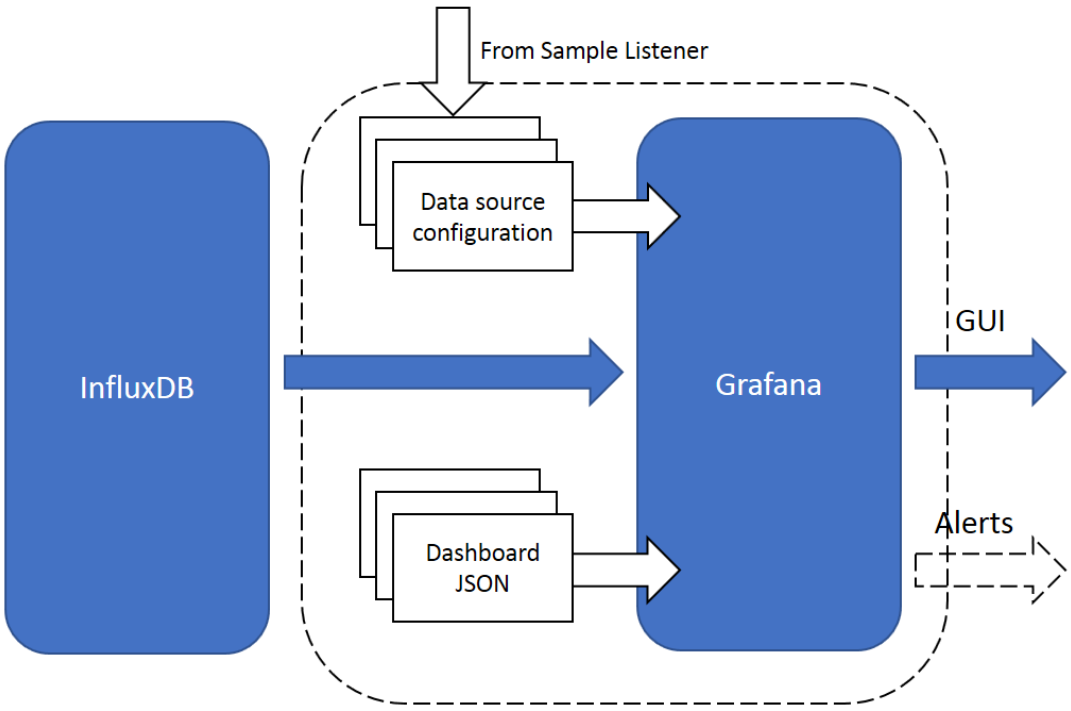

Figure 3.6 Grafana configuration

Dashboards are stored in a JSON format and they can either be made in Grafana after the system has started up or they can be imported during the startup procedure as JSON files. This feature further extends the possibilities to automate the system setup procedure.

3.4 System Setup

The components of Data Visualizer require several configurations before they can be started. Some of these configurations are also dependent on other components of the system. The setup steps and their correct execution order are introduced in chapter 3.4.1 while chapter 3.4.2 focuses on automation of these steps.

3.4.1 Manual Setup

The correct setup procedure for the system is presented in figure 3.7. It is assumed that the person setting up the system has access to InfluxDB and Grafana binaries, and knows how to configure and set them up. Source codes for the sample listener and a pipeline data parser along with example dashboard JSON files for Grafana are provided by the Data Visualizer system.

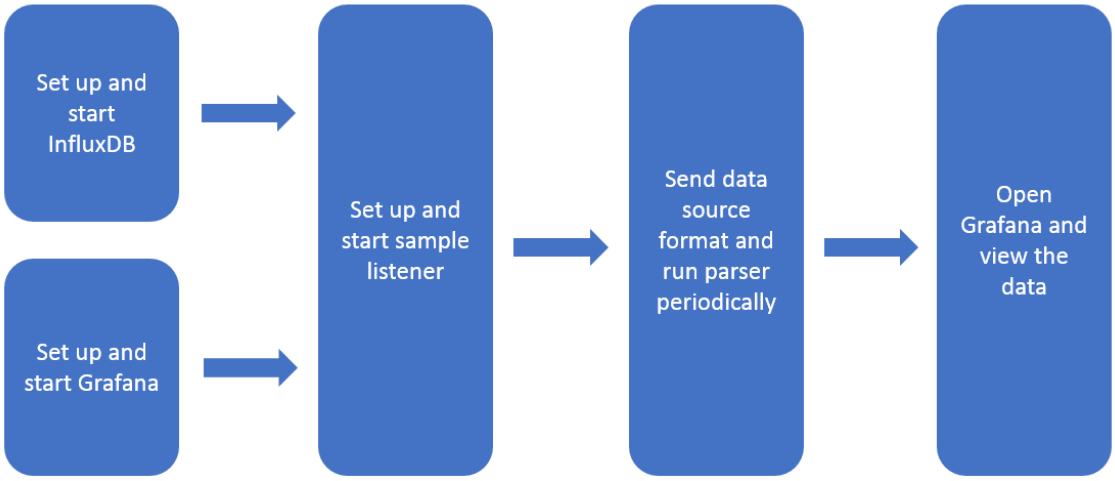

Figure 3.7 Setup process

After InfluxDB and Grafana have been set up, sample listener can be started. The URL address of InfluxDB and Grafana as well as any login information required to make administrative changes to these systems must be known by the listener. During the startup, the listener will automatically make required changes to InfluxDB and Grafana.

Existing dashboards can be added to Grafana either during the setup phase by using a custom Grafana configuration or after the setup by using the dashboard import feature. New dashboards can also be created by using the data source definitions provided by the sample listener.

When the sample listener has completed all system configurations, the data source format can be sent to the listener and the pipeline data parser can be started to gather pipeline data. The parsing can be either done manually or automated with a task scheduler. Finally, the gathered data can be viewed in Grafana dashboards.

3.4.2 Docker and Docker Compose

Since the manual setup has several steps and requires multiple components to be configured, increasing automation becomes necessary to simplify the setup process. It also decreases the possibilities of human errors and the amount of InfluxDB and Grafana specific knowledge needed to set up the system.

Docker is a virtualization software that provides the ability to create closed environments named *containers* for specific applications. Creating a container simplifies the automation of the various setup procedures since it removes any OS specific dependencies. It also helps to productize the system to a package that is simple to set up experiment with.

Docker Compose provides automation tools for creating multi-container Docker environments. It automates the creation of each container and creates the network mapping between each container and the host system. Dividing the system to separate containers also simplifies the development and configuration of each system component.

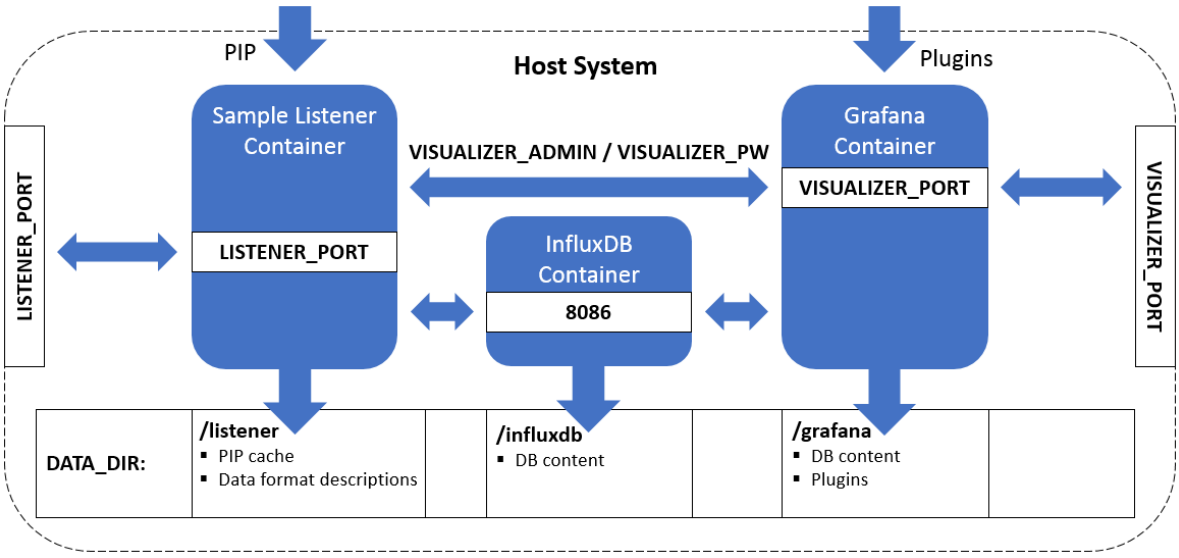

Figure 3.8 Docker Compose system

Overview of the Data Visualizer system created with Docker Compose is presented in figure 3.8. The containers share a common data volume mounted at the host system path defined in variable DATA_DIR for all data that should persist between different builds. For sample listener, this means third party Python modules downloaded with the PIP package management tool as well as the data format descriptions used to configure the system. InfluxDB and Grafana will store their database content. Grafana will also store any downloaded plugins.

Grafana and sample listener are connected to the host system by exposing the ports defined in variables LISTENER_PORT and VISUALIZER_PORT. The port number of InfluxDB is hard coded to 8086 since it is only used for internal communication between the containers. Additionally, communication between Grafana and sample listener requires the use of the Grafana administrator username and password defined in variables VISUALIZER_ADMIN and VISUALIZER_PW.

Each container is created using a Dockerfile that contains a sequence of instructions needed to set up the system. The Docker Hub website is a public library containing container definitions for various widely used applications, including Python, InfluxDB and Grafana. These containers are used as the basis for the separate components of Data Visualizer. The Dockerfiles of the containers can be found in appendix B.

The network and file system mapping of the containers is handled by Docker Compose. This process is defined by a YML file containing the system description in a format defined by Docker Compose. Additionally, the description may be parametrized by listing any required variables and their values in a file named ".env" in the directory the Docker Compose setup is executed. The Docker Compose YAML file and the default parameter definitions for Data Visualizer can be found in appendix C.

Figure 3.9 Setup process using Docker Compose

With the use of the Docker Compose environment, the component setup parts of figure 3.7 are combined and simplified to the phases described in figure 3.9. The build.sh script is used to further simplify the build process for users unfamiliar with the Docker environment. By giving the users premade parsers, data source formats, and Grafana dashboards, the setup instructions can be simplified to a compact list of terminal commands.

Furthermore, any built Docker images can be shared across multiple environments, which allows the developer of the system to manage the images while the users can only define the basic run-time configuration and use up command of Docker Compose. Image distribution systems such as the public Docker Hub repository can be set up internally within organizations for simple access to the images.

3.5 System Extensions

The components of Data Visualizer system offer multiple interfaces to further extend the system functionalities. The sample listener component offers tools for handling written data while Grafana offers links to various reports and automatic generation of alerts from the implemented panels.

This section describes additional tools created to increase the usability of the system as well as the changes made to the implementation of the Data Visualizer components to enable the development of these tools. Currently there are two additional tools available: a Grafana picture library to store the most important reports and a sample backup service to create a snapshot of the sample content inside the system storage.

3.5.1 Grafana Picture Library

Grafana is a powerful tool for discovering and managing efficient ways to present various sources data. However, for a user that repeatedly views the same data to quickly see a current overall status, the interface may become cumbersome to use. Implementation of a picture extraction and archiving tool simplifies the user interface and offers quick access to status reports created in Grafana. The basic look of the user interface is presented in figure 3.10.

Figure 3.10 Grafana snapshot library

By utilizing Docker to package the snapshot library server, multiple server instances can be set up for various related measurements with little effort. When provided with the URL addresses to the Grafana panels of each measurement, the server can automatically generate and set up the web page and begin archiving reports. This provides a straightforward way to store and access the most important reports while eliminating the requirement of having to find the same data in Grafana.

3.5.2 Sample Backup

The need for a sample backup tool became apparent during the development of Data Visualizer as parsing and sending the same data repeatedly was time consuming. Implementation of new endpoints to the sample listener for querying format and sample XML data removed the need for repeated parsing. Additionally, a backup scheduler was implemented to periodically query for sample data, preventing data loss from the system. The data flow of this additional component is presented in figure 3.11.

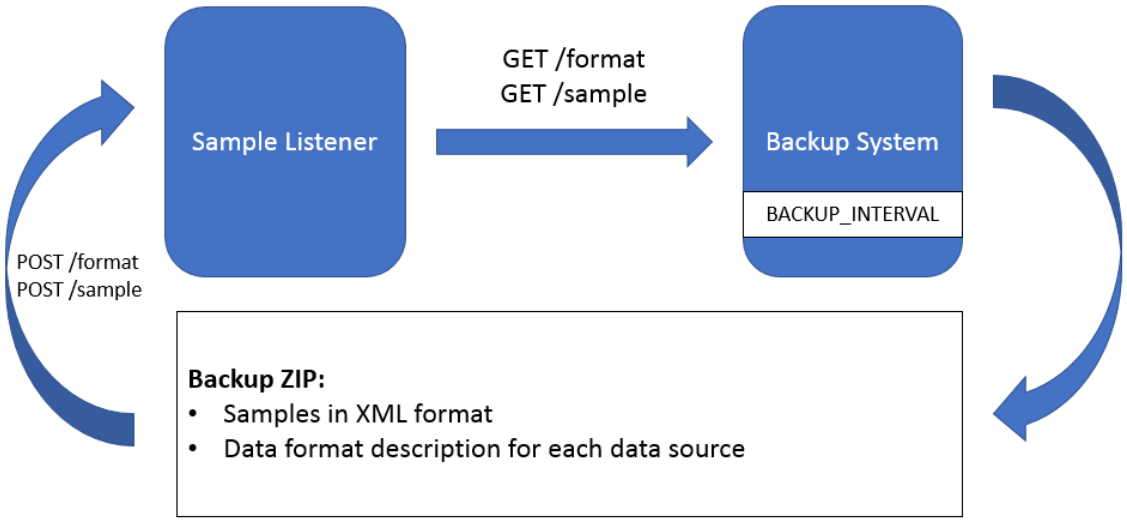

Figure 3.11 Backup and restore process

The backup scheduler queries data source format descriptions from the */format* endpoint of the sample listener and then continues to query sample XML for each received format. All the received data is stored under a directory and compressed to a ZIP file. The user is then free to decide how to manage these system snapshots. A snapshot can be restored simply by uncompressing the ZIP file and sending each format description to the sample listener followed by the sample XML files of that format.

4. EVALUATION

The Data Visualizer system described in chapter 3 was taken into use as a part of the verification process described in chapter 1. It was used to collect and visualize data mainly from two sources: Jenkins and Robot Framework. It helped to identify previously unclear problems in the verification process and the use of these two tools.

The following subchapters will present the two use cases from data source analysis to building the data parser for Data Visualizer and finally to presenting the data in Grafana. Afterwards, the written Python code is analyzed for both the Data Visualizer system and the previously presented use cases. Finally, some general analysis will be given for the entire work.

4.1 Use Case: Jenkins

Jenkins is a popular, open source automation server that is often used for implementation of automated Continuous Delivery systems. It allows the user to wrap the functionality of repeatedly performed tasks into jobs that can be executed either manually or automatically with a scheduler. [31]

Development of CD Pipelines for systems with complex build procedures is further simplified with use of the Jenkins Pipeline plugin. It introduces a Groovy based pipeline description language where the pipeline functionality is divided into named stages. Each stage executes a sequence of steps defined by the language, such as command line execution. The number and functionalities of the pipeline steps can be extended with use of additional Jenkins plugins.

4.1.1 Data Analysis: Building the Jenkins Format

The amount of collectable data in a Jenkins based CD system is mostly tied to the number and type of installed plugins. This use case covers two separate sources of data for Jenkins and the Pipeline plugin: the REST APIs and the Jenkins home directory files. Separate data parsers were written for both sources.

The Jenkins REST API can be accessed from any URL address of a Jenkins instance by appending API and data type segments to the URL path. Additionally, the Pipeline plugin has a separate REST API where similar data can be found on pipeline, stage, and step levels. While the Jenkins interface can send data in multiple formats, the Pipeline plugin API is restricted to a JSON based format. Therefore, the JSON format is used for both interfaces.

A simple data source format to encompass both Jenkins job and pipeline data is proposed in appendix D. It divides the data into three samples: build, stage, and step. Each of these samples contain duration integer as measurement with name and result strings as identifiers.

4.1.2 Data Formatting: Writing the Jenkins Parser

The first source for the data defined in appendix D is the Jenkins REST API. It can be used to gather the required data for each build sample. Figure 4.1 presents a general structure for the data received from the Jenkins REST API when JSON data type is selected.

```
{   "_class": ...,
    "jobs": [
        {   "_class": ...,
            "name": "job1",
            "builds": [
                {   "_class": ...,
                    "number": "1",
                    "timestamp": "1514764800000",
                    "duration": "100",
                    "result": "SUCCESS",
                    ...
                },
            ]
        },
        {   "name": "job2",
            ...
        },
        ...
        {   "name": "jobN",
            ...
        },
    ],
}
```

Figure 4.1 Jenkins JSON REST API information

To create the stage and step samples within each build, the separate REST API of the Pipeline plugin must be used. The job name and build number sections can be used to construct the URL address for the Pipeline plugin REST API. The address consists of the Jenkins address continued with path */job/job_name/build_number/wfapi/describe*, where *job_name* and *build_number* are replaced with the job name and build number respectively. Example data returned from this address is presented in figure 4.2.

```
{   "_links": {
        "self": { "href": "/job/job1/1/wfapi/describe" },
        "changesets": { "href": "/job/job1/1/wfapi/changesets" }
    },
    "id": "1",
    "name": "#1",
    "status": "SUCCESS",
    "startTimeMillis": 1514764800000,
    "endTimeMillis": 1514764801000,
    "durationMillis": 1000,
    "queueDurationMillis": 0,
    "pauseDurationMillis": 0,
    "stages": [
        { "_links": {
              "self": {
                  "href": "/job/job1/1/execution/node/4/wfapi/describe"
              }
          },
          "id": "4",
          "name": "Stage 1",
          "execNode": "",
          "status": "SUCCESS",
          "startTimeMillis": 1514764800000,
          "durationMillis": 100,
          "pauseDurationMillis": 0
        }
        ...
    ]
}
```


Figure 4.2 Pipeline plugin build information

The previously described API address only returns data on stage level. To create the sample elements for each step, the link path inside each stage must be appended to the Jenkins address. Figure 4.3 describes the data returned from this address.

```
{   "_links": {
        "self": {
            "href": "/job/job1/1/execution/node/4/wfapi/describe"
        }
    },
    "id": "4",
    "name": "Stage 1",
    "execNode": "",
    "status": "SUCCESS",
    "startTimeMillis": 1514764800000,
    "durationMillis": 100,
    "pauseDurationMillis": 0,
    "stageFlowNodes": [
        {   "_links": {
                "self": {
                    "href": "/job/job1/1/execution/node/5/wfapi/describe"
                },
                "log": {
                    "href": "/job/job1/1/execution/node/5/wfapi/log"
                }
            },
            "id": "5",
            "name": "Shell Script",
            "execNode": "",
            "status": "SUCCESS",
            "startTimeMillis": 1514764800000,
            "durationMillis": 10,
            "pauseDurationMillis": 0,
            "parentNodes": ["4"]
        },
        ...
    ]
}
```

Figure 4.3 Pipeline plugin stage information

Since parsing the Jenkins data through the REST APIs requires multiple requests to various addresses, the process may be significantly slowed down depending on the current API load. The data received on pipeline step level is also rather basic. For example, the execution of a shell script only has the node name “Shell Script” available to be used as the step name, causing individual shell script steps to become indistinguishable.

The Jenkins directory structure described in figure 4.4 offers an alternative source for the same data in XML form. This eliminates the requirement for multiple HTTP requests and allows the parser to access larger amount of build, stage and step data. However, the large amount of data in numerous XML files may make this approach slower than the previous alternative depending on the execution environment and the content of the parsed job data. For example, in the environment the parsers were first taken into use, the parsing times of the Jenkins XML parser were approximately twice as long when compared to the REST alternative. Extensive testing on the performance difference between the two implementations has not yet been done.

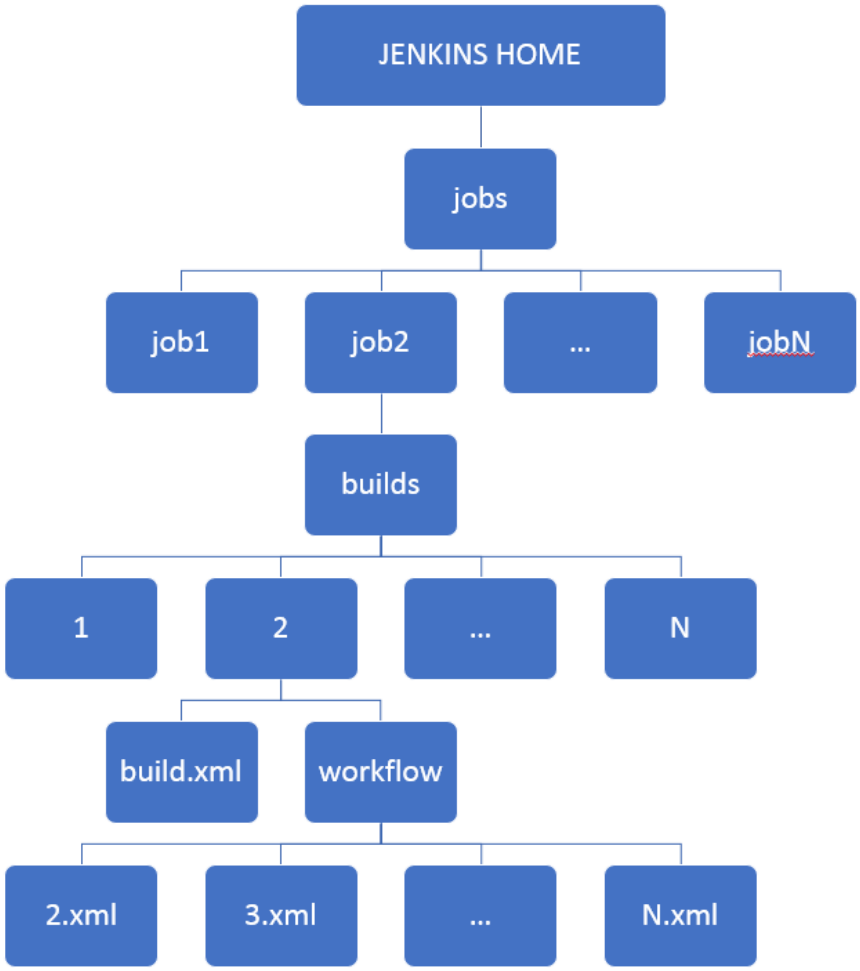

Figure 4.4 Directory structure of Jenkins with Pipeline plugin

The coding effort of the two parser implementations was minimized by reusing code for the user interface and most of the data parsing. The reusable parts were written into a base class while extraction of the job data was implemented in two separate subclasses. The time required from a single person to create the Jenkins REST parser was approximately one day while the more complex XML parser took roughly two days to yield the first functional version.

4.1.3 Visualization: Building the Jenkins Dashboard

Duration and result information for job builds alone yield numerous possibilities for visualizing the system status. When combined with similar data for each pipeline stage, the system can be efficiently used to track down problems in the pipeline workflow. This chapter will go through some basic visualizations for Jenkins jobs.

Figure 4.5 Job duration visualization

The first obvious visualization for Jenkins job data is to display the job duration over time. Since jobs are usually expected to behave differently during succeeding and failing runs, it is beneficial to separate these two cases as separate lines. This separation can also be used to analyze the behavior of the job in more detail.

Figure 4.5 presents such visualization for example data. In this example a rising trend in the duration of the successful job runs is clearly visible despite the two steep declines. Depending on the nature of the job, this behavior could have a wide variety of causes. It could occur as intended as more functionalities are added to the job or it could be a design flaw that accumulates over time. Both the rising trend and the steep declines are things that should be thoroughly investigated.

The duration of the failing runs also frequently rises far above the duration of the succeeding runs. For example, in the case of failures caused by timeouts, the timeout restrictions inside the job could be investigated to minimize the time spent on waiting for the execution to fail. Failing on timeouts earlier and restarting the job might be faster than waiting for the system to recover in certain cases.

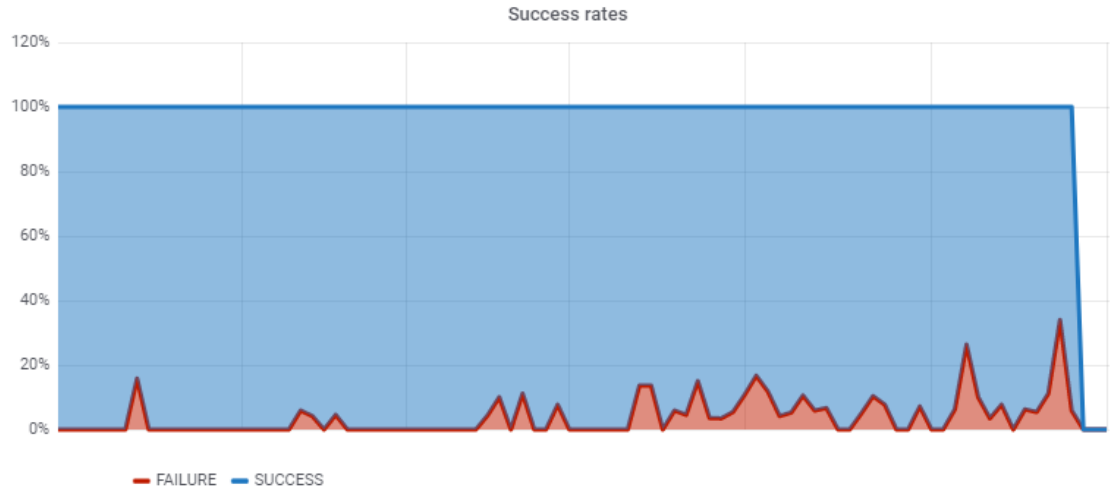

Figure 4.6 Job success rate visualization

Another straightforward way to monitor Jenkins jobs is to view the rate of success and failure over time. In a continuous delivery system, working towards high success rates is important as the scope and size of software issues are expected to be kept small and manageable.

Figure 4.6 shows a simple visualization of success rates for the same data as in figure 4.5. The overall failure rate is low, although a slight increasing trend can be seen. The success rate view itself does not offer much insight on the reasons of this rise. It can be used as a tool to check whether further investigation on other visualizations of the system is needed.

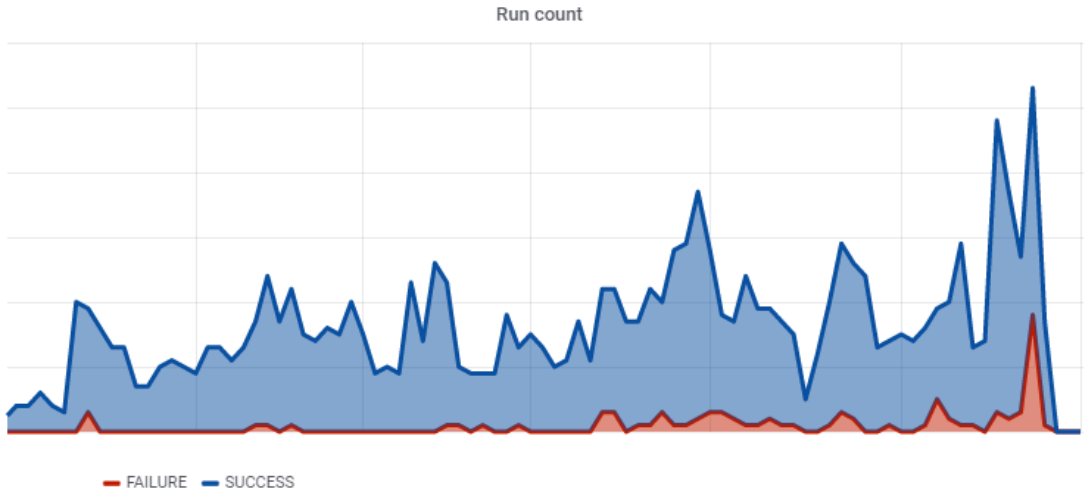

Figure 4.7 Number of executions for a single job per day

Counting job executions gives some information about the utilization of the job. It can be used to determine resource requirements for the system running the job. It might also reveal some effects of the utilization rates on the success rates. In the example view of figure 4.7, a slight increasing trend on the job utilization can be seen along with the increase in failure rates presented in figure 4.6.

The visualizations introduced in this chapter are only few simple examples for the use of Grafana. Differences, cumulative sums, combinations of duration an execution count and many other visualization techniques can be used to further extend the possibilities for job analysis.

4.2 Use Case: Robot Framework

Robot Framework is a widely used test automation tool for acceptance testing. It is often used as a verification tool in various CI pipelines and many software development tools offer some form of integration with it. For example, the Jenkins environment described in chapter 4.2 allows integration of Robot Framework testing and reporting through plugins. [32]

4.2.1 Data Analysis: Building the Robot Format

Robot Framework generates test result reports to an XML formatted output file. This file contains various logging information for each executed suite, test, and keyword as well as general statistics for the entire test run. Figure 4.8 presents the structure for some of the timestamp and duration information found in the output file.

```
<robot ...>
    <suite name=”mainsuite”>
        <suite name=”suite1”>
            <test name=”test1”>
                <kw>
                    <status/>
                </kw>
                <kw>...</kw>
                ...
                <kw>...</kw>
                <status status=”PASS” startTime=”” endtime=”” />
            </test>
            <test name=”test2”>...</test>
            ...
            <test name=”test3”>...</test>
            <status status=”PASS” startTime=”” endtime=””/>
        </suite>
        <suite name=”suite2”>...</suite>
        ...
        <suite name=”suite3”>...</suite>
        <status status=”PASS” startTime=”” endtime=””/>
    <suite>
    ...
</robot>
```

Figure 4.8 Robot Framework output XML

Based on the XML structure of figure 4.8, a sample format for Robot framework result and duration information is proposed in appendix E. The format defines a sample hierarchy of a *mainsuite* containing *suites* that further contain either other *suites* or *tests*. Keywords are not included as separate samples since they usually do not contain information that would stay relevant over extended periods of time.

4.2.2 Data Formatting: Writing the Robot Parser

The base of the output XML parser can be built around the structure described in figure 4.8 with little effort. The suite and test elements can be mapped directly to the corresponding sample elements and the proceeding status elements can be used to fill the remaining attributes. The parsers can then be built on top of this implementation for various sources of output XML files.

To tie the use case of Robot Framework together with the previous use case of Jenkins, a parser for the Jenkins Robot Framework plugin was implemented. This parser was used to fetch output XML files from the logs of Jenkins jobs using the Robot Framework plugin and parse them using the parser base implementation.

4.2.3 Visualization: Building the Robot Dashboard

Visualization possibilities in Robot Framework are mostly identical to the Jenkins job visualizations discussed in chapter 4.2.3. Duration trends, success rates and counts of executed suites and tests can be monitored with similar visualizations revealing similar problems.

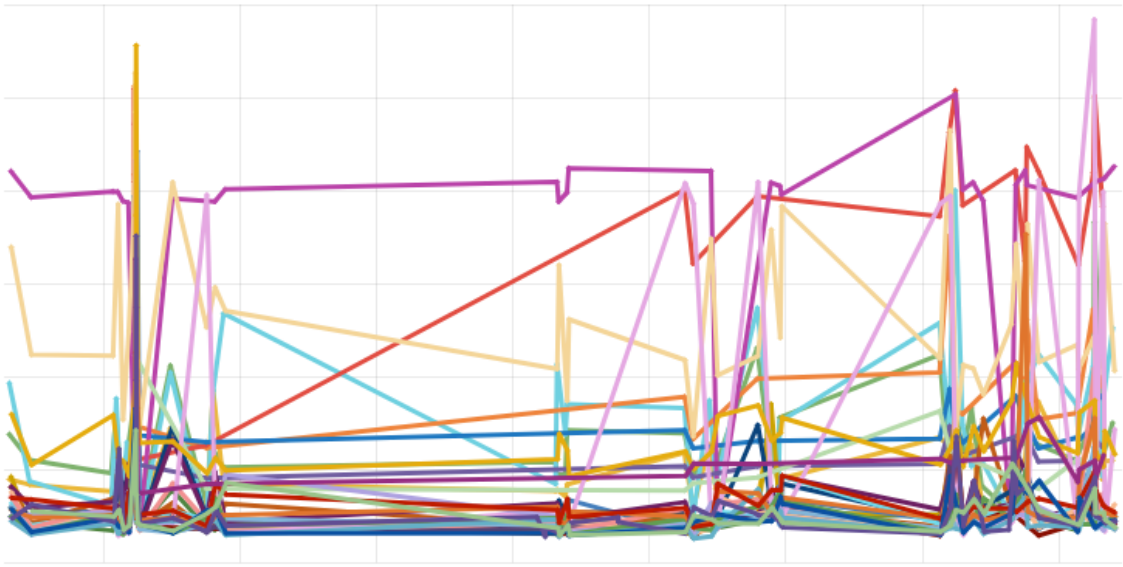

Figure 4.9 Test durations for a single suite

During the development of various visualizations for Robot Framework data, the importance of clear visualizations became apparent. In cases with suites containing many tests, displaying all duration trends in one panel became challenging, as demonstrated in figure 4.9. However, since the panels in Grafana have interactive features that allow displaying of selected lines only, this problem becomes simple to deal with. A single line from figure 4.9 is displayed in figure 4.10 using this feature.

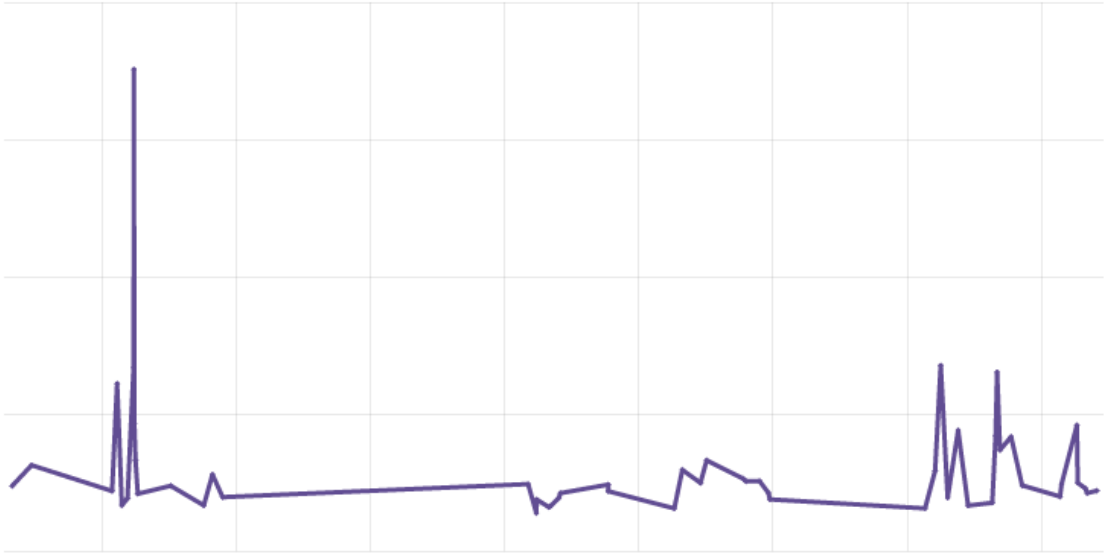

Figure 4.10 Duration of a single test

Another problem with the previously presented visualizations is the inability to monitor parallel execution. In the case of Robot Framework executing suites and tests in parallel whenever possible would greatly increase testing speed and productivity. Visualizing this behavior is also important as it helps to reveal dependency issues in parallel tests.

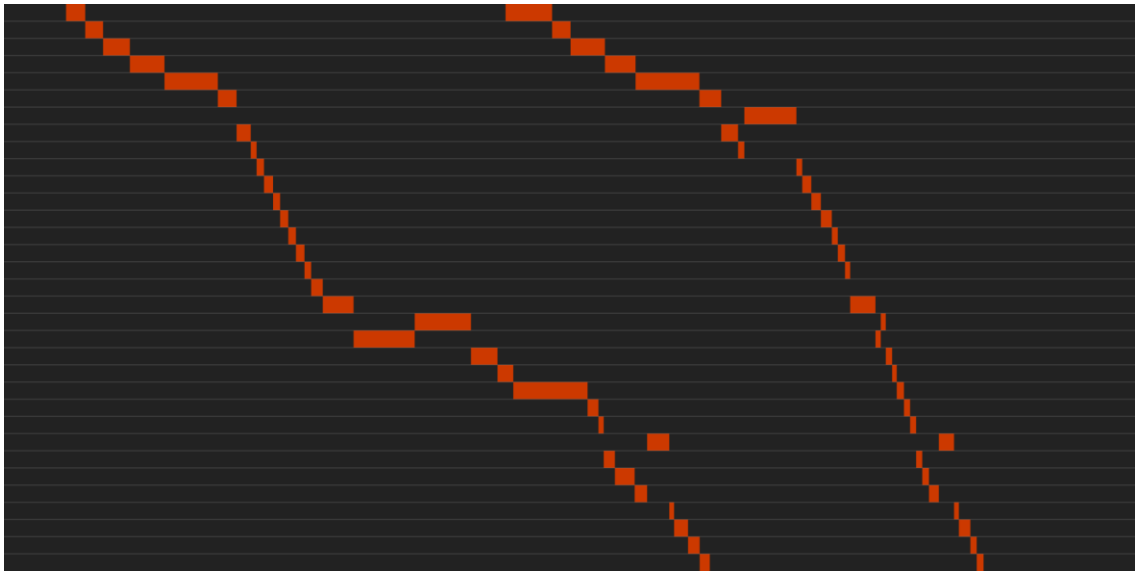

Figure 4.11 Test execution events

A discrete event viewer plugin for Grafana offered clear representation of test execution by utilizing the starting and ending timestamps of each test. By displaying the tests of figure 4.9 in this visualization, the picture shown in figure 4.11 was received. After analyzing the results, it was clear that the suites were properly utilizing parallel execution, but the test cases themselves were all ran sequentially. This discovery presented significant potential for reduction of test execution times.

4.3 Work Effort

By utilizing open source projects such as InfluxDB and Grafana as well as the available XML parsing and HTTP server libraries available for Python, the coding effort required for the development of Data Visualizer was simple enough to be managed by a single person. Historically, when open source content was not as widely available and possibly of low quality, similar projects would have required either vastly more effort or costly third-party tools.

The visualization service was built from ground up and went through a complete software development process from planning of the service requirements to productization of the Data Visualizer system. Documentation was done on code, component, and system levels along with simple user guides, a wiki page, and presentation material to get inexperienced users introduced to the system. A continuous integration process was also established with automated unit testing for the Python code and a review process for every committed change to the system.

The proceeding subchapters will further analyze the self-written Python code used in development of the sample listener and parser components. Other code, such as Dockerfiles and their build and startup scripts are not included.

4.3.1 Written Code

All the developed components were written in Python with the distinction of the data parsers being compatible with both Python versions 2.7 and 3.5, while the sample listener was written only for Python 3.5. This decision was made to increase combability of the data parsers with a wider range of systems while focusing on performance in the system running the sample listener.

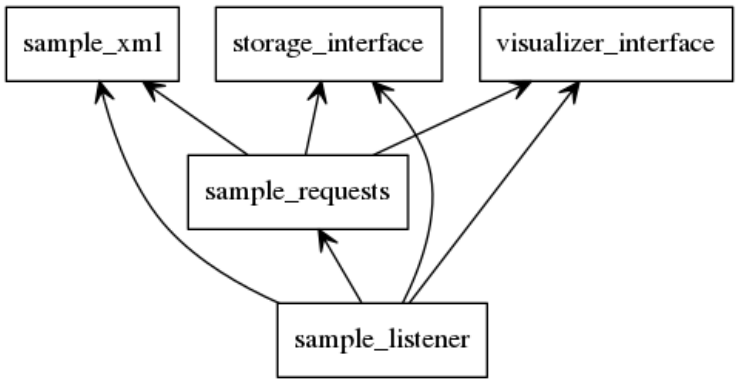

Figure 4.12 Module diagram of sample listener

The module and class diagrams presented in figures 4.12 and 4.13 are generated for the sample listener using the *pyreverse* tool. The tool can be used to reverse engineer given Python modules, which gives a rough picture of the overall code structure. It provided as a part of the *pylint* code analyzer which can be further used to analyze the quality of the written code.

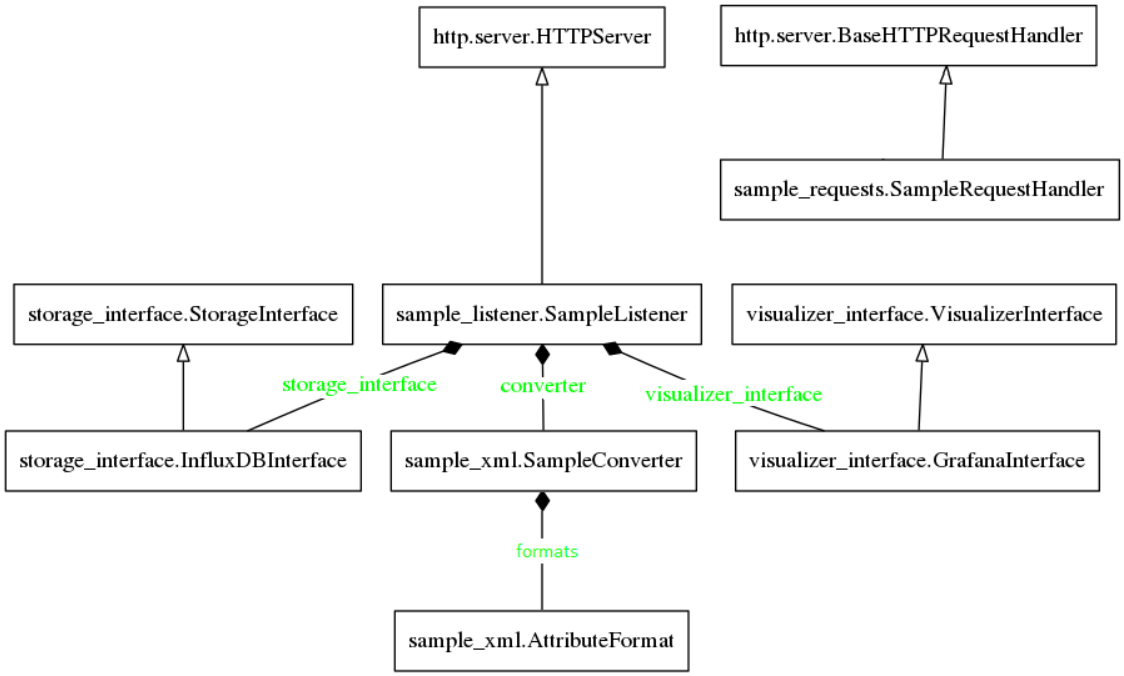

Figure 4.13 Class diagram of sample listener

As seen in figure 4.13 the sample listener consists of three main components: the storage interface, the sample converter, and the visualizer interface. The methods and attributes of storage and visualizer interfaces are defined separately while their implementations are written inside the InfluxDB and Grafana specific interfaces.

Sample converter validates and stores the given data source format descriptions in a dictionary structure containing validator classes for each sample attribute. This structure is then used to validate given samples and extract their data.

Implementation of the sample listener is based on the HTTP server implementation of the built-in *http* library where the request handling is separated to a subclassable handler class. Whenever the sample listener receives an HTTP request, it creates a new instance of the sample request handler that uses the three previously described components to handle the request.

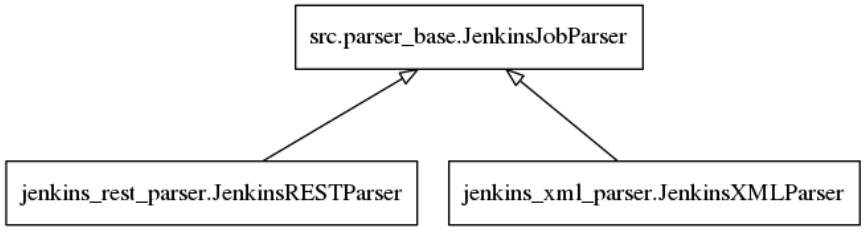

Figure 4.14 Class diagram of the Jenkins parsers

As described in chapter 4.1.2, the parsing of Jenkins data is split into two separate parsers: the Jenkins REST API parser and the Jenkins home directory parser. Both parsers are based on a base parser implementation that contains the common code. Class diagram for the Jenkins parsers is presented in figure 4.14.

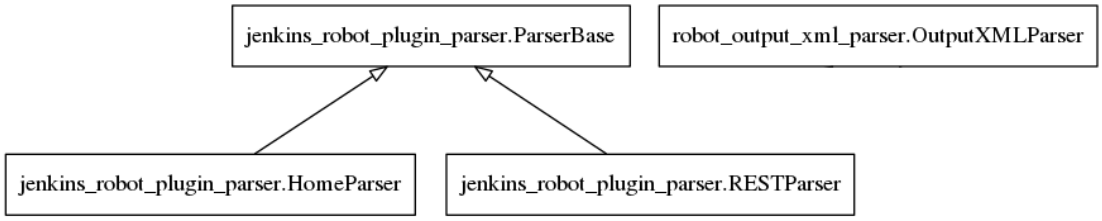

Figure 4.15 Class diagram of the Robot Framework plugin parsers

The Robot Framework output XML parser is implemented as a single class based on the input data described in chapter 4.2.1. The parser for a single output XML are used as the basis for derivative parser implementations where multiple output XML files are read from a specific data source. As both derivative parsers are based on the Jenkins Robot Framework plugin, their implementations can be connected through a subclassable base parser similarly to the Jenkins data parsers. Class diagrams for the Robot Framework parser are presented in figure 4.15.

The overall amount of the written code was large but manageable for a single person. It dealt with a highly diverse set of problems from parsing of time series data to creation of automatic database configurations and handling of large server loads. Many optimizations the flow of data from the source to InfluxDB were required to create a functional system.

4.3.2 Unit Testing of the Visualizer

Python *virtualenv* and *nose* tools provide simple methods for running unit tests in a portable environment that fulfills all the requirements of the written code and generating coverage reports based on the results. After the unit tests have been written, they can be wrapped together in a single script that sets up the *virtualenv* with only required modules installed and scans the project directories for test cases using *nose*. This greatly simplifies moving the development of the project from one environment to another and enables the use of continuous integration.

Module ↓	statements	missing	excluded	coverage
data_formatting/parsers/jenkins/jenkins_rest_parser.py	156	6	0	96%
data_formatting/parsers/jenkins/jenkins_xml_parser.py	180	5	0	97%
data_formatting/parsers/jenkins/src/jenkins_workflow.py	260	0	0	100%
data_formatting/parsers/jenkins/src/parser_base.py	127	5	0	96%
data_formatting/parsers/jenkins/tests/jenkins_rest_parser_tests.py	120	4	0	97%
data_formatting/parsers/jenkins/tests/jenkins_workflow_node_tests.py	53	2	0	96%
data_formatting/parsers/jenkins/tests/jenkins_workflow_parser_tests.py	115	2	0	98%
data_formatting/parsers/jenkins/tests/jenkins_xml_parser_tests.py	206	3	0	99%
data_formatting/parsers/jenkins/tests/parser_base_tests.py	113	4	0	96%
data_formatting/parsers/robot/jenkins_robot_plugin_parser.py	94	6	0	94%
data_formatting/parsers/robot/robot_output_xml_parser.py	243	5	0	98%
data_formatting/parsers/robot/tests/jenkins_robot_plugin_parser_tests.py	128	5	0	96%
data_formatting/parsers/robot/tests/robot_output_xml_parser_tests.py	93	4	0	96%
storage/listener/src/sample_listener.py	102	2	0	98%
storage/listener/src/sample_requests.py	287	0	0	100%
storage/listener/src/sample_xml.py	312	0	0	100%
storage/listener/src/storage_interface.py	231	0	0	100%
storage/listener/src/visualizer_interface.py	93	0	0	100%
storage/listener/tests/attribute_format_tests.py	61	2	0	97%
storage/listener/tests/common_functions.py	55	1	0	98%
storage/listener/tests/format_schema_tests.py	141	9	0	94%
storage/listener/tests/grafana_interface_tests.py	122	2	0	98%
storage/listener/tests/influxdb_interface_tests.py	201	2	0	99%
storage/listener/tests/sample_converter_tests.py	133	3	0	98%
storage/listener/tests/sample_listener_tests.py	106	2	0	98%
storage/listener/tests/sample_requests_tests.py	190	2	0	99%
Total	**3922**	**76**	**0**	**98%**

Figure 4.16 Unit test coverage report for Python 3.5 compatible code

Figures 4.16 and 4.17 contain the code coverage reports generated by nose runs for Python versions 3.5 and 2.7. The reports show that the total amount of code was divided almost evenly between the system code and the test cases. The coverage percentage itself shows that nearly every line of system has been run by the test cases, but does not provide further information about the quality of the tests.

Module ↓	statements	missing	excluded	coverage
data_formatting/parsers/jenkins/jenkins_rest_parser.py	156	4	0	97%
data_formatting/parsers/jenkins/jenkins_xml_parser.py	180	5	0	97%
data_formatting/parsers/jenkins/src/jenkins_workflow.py	260	0	0	100%
data_formatting/parsers/jenkins/src/parser_base.py	127	3	0	98%
data_formatting/parsers/jenkins/tests/jenkins_rest_parser_tests.py	120	2	0	98%
data_formatting/parsers/jenkins/tests/jenkins_workflow_node_tests.py	53	2	0	96%
data_formatting/parsers/jenkins/tests/jenkins_workflow_parser_tests.py	115	2	0	98%
data_formatting/parsers/jenkins/tests/jenkins_xml_parser_tests.py	206	4	0	98%
data_formatting/parsers/jenkins/tests/parser_base_tests.py	113	2	0	98%
data_formatting/parsers/robot/jenkins_robot_plugin_parser.py	94	3	0	97%
data_formatting/parsers/robot/robot_output_xml_parser.py	243	3	0	99%
data_formatting/parsers/robot/tests/jenkins_robot_plugin_parser_tests.py	128	2	0	98%
data_formatting/parsers/robot/tests/robot_output_xml_parser_tests.py	93	2	0	98%
Total	**1888**	**34**	**0**	**98%**

Figure 4.17 Unit test coverage report for Python 2.7 compatible code

The effort of writing the unit test code was approximately equal to writing the system code itself. After the testing framework was set up, the effort required to commit changes to the system was decreased significantly. The coverage reports generated by *nose* also helped to quickly identify the requirements for new test cases when a new feature was implemented.

4.4 General Analysis

This chapter analyzes the design of Data Visualizer from the perspectives of performance, maintainability, reusability, portability, and further development. The analysis is given for the complete system as well as each of its main components. Additionally, some alternative approaches to the ones currently implemented in Data Visualizer are given.

4.4.1 Performance

Performance optimization plays a vital role in each phase of the system. Any of the system components could severely bottleneck the flow of data when not optimized for performance. Which component causes the bottleneck depends mainly on three main factors: properties of the used data sources, properties of the defined sample formats, and the number of users viewing the data.

The properties of a data sources affect the performance of both data formatting and storage phases. As the size of a data source increases, the likelihood of the data parser implementation becoming the system bottleneck increases. Similarly, the complexity of the data source affects the time spent on parsing the data. From the performance perspective it is therefore important to favor data sources that require less parsing of unwanted data complex operations to form a sample.

The defined sample formats may have significant effects on the performance of the underlying storage container inside the storage system. In the case of InfluxDB it is important to only pick sample identifiers that have limited and preferably small number of different values. This is something that should be considered first during the creation of the format description.

When the number of users viewing the gathered data increases, the performance optimization of the visualization phase becomes more important. Grafana dashboards should be designed in a way that minimizes the number of data source queries required to view the data and the queries themselves should not handle massive amounts of data.

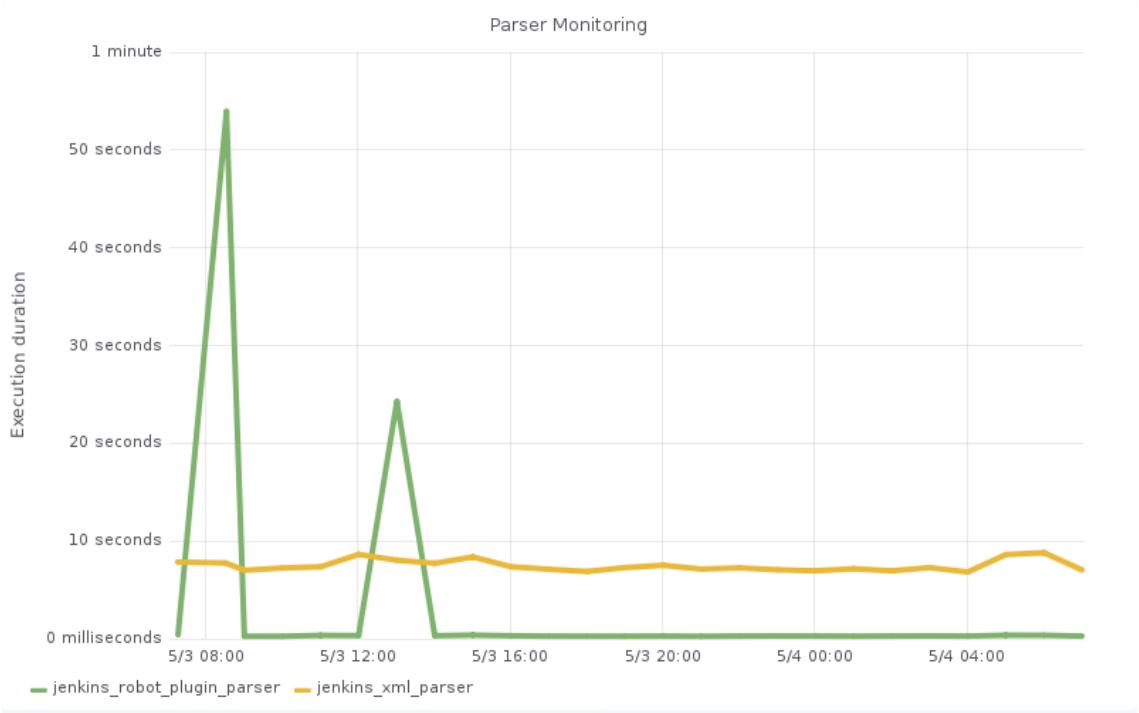

Figure 4.18 Duration of data parser executions

From the perspective of the entire system all the before mentioned factors should be monitored to determine the state of the system performance. A straightforward way to perform this would be to utilize the system to monitor itself. For example, the duration of the data parser executions could be measured as demonstrated in figure 4.18. As the figure shows, some optimization might be necessary in the Jenkins Robot Framework plugin parser since the parsing times can rise up to a minute when new test reports are available.

4.4.2 Maintainability

Since the data sent to the Data Visualizer system is often stored and viewed over extended periods of time, making changes to the system core components becomes challenging. Specifically, changes to the sample listener component that affect the system configurations or the conversion of sample data will require a clean setup of the system.

Another critical change to the system is the redefining of a data source format description. Unlike the changes to the system core components, these changes do not require a clean setup for the system. Instead, the previously written sample data must be either removed along with the old data source format, separated from the new format version with a different format name, or migrated to the new format by rewriting the samples.

By enforcing the generic sample format introduced in chapter 3.1.1, the effects of these changes are limited to the environment of the sample listener while the functionalities of the data parser components remain the same. Similarly, when the format description of a data source is changed, the only effect on the data parser components is the implementation of any newly added or removed sample attributes.

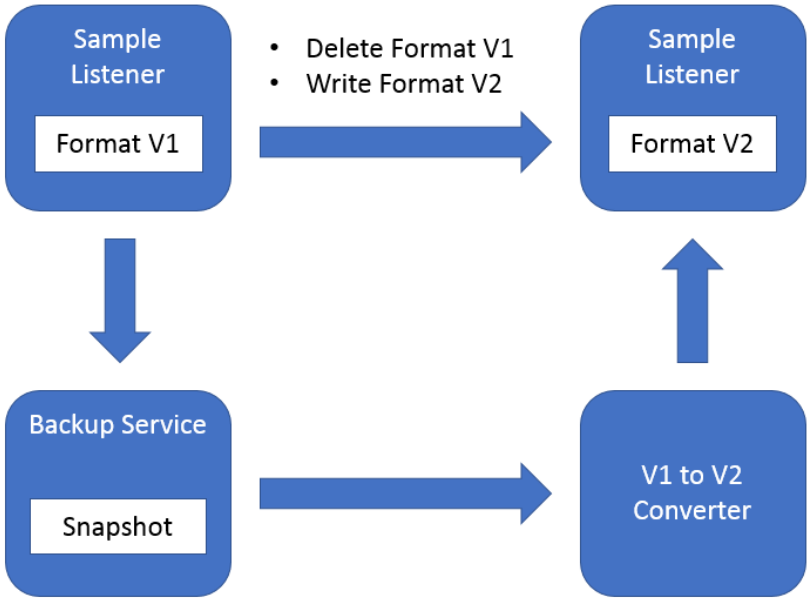

Figure 4.19 Format update migration

When a critical change occurs on the system, the sample backup service described in chapter 3.5.2 will simplify the required procedures to rewrite the sample data as seen in figure 4.19. After the latest backup cycle, the required change can be applied and the previously written sample data will remain available in the backup service. These samples can be simply sent to the sample listener along with their existing formats or migrated to a new format.

4.4.3 Reusability

As Data Visualizer relies mostly on third party implementations for the data container and visualization, it is important to design the system components as reusable as possible in case one of these implementations requires a replacement. In addition, the data parser components benefit greatly from added reusability when writing alternative parser implementations for a single data source format. This is demonstrated with the Jenkins parser implementations described in chapter 4.1.2.

The possible replacements of InfluxDB and Grafana are addressed with the implementations of generic interfaces for the storage container and the visualization component. When a more suitable implementation for either of these components is discovered, only rewriting the interface implementation for the replacement component is required.

In the sample listener component, the inclusion of format version information in the data source format XML increases the reusability of existing sample formats. However, it also increases the required maintenance effort when managing multiple format versions. The reuse of formats with old versions can be supported until most of the user base has migrated their data to newer versions.

4.4.4 Portability

Data Visualizer is designed to be used in various pipelines by users with varying knowledge of the system. Therefore, its setup and usage should be simple to minimize the need for direct user support. The system is provided with detailed usage documentation and examples. A wiki page and presentation material are also provided to introduce the key aspects of the system to the user and link it to the monitored systems.

Utilizing the Docker environment simplifies the setup process significantly. To set up a monitoring system with predefined data parsers and Grafana dashboards, the user must fill out the Docker Compose environment configuration, run a single build script, start the system with the *up* command provided by Docker Compose, and write the data source formats by sending them to the */format* endpoint of the sample listener component.

4.4.5 Further Development

All three system components offer HTTP APIs for possible system extensions, two of which were presented in chapter 3.5. Since the Data Visualizer framework attempts to be as modular as possible, it would be natural to continue the development with even more modular components. Implementation of an alerting and anomaly detection system is used as an example to describe the different approaches available when creating extensions to the system.

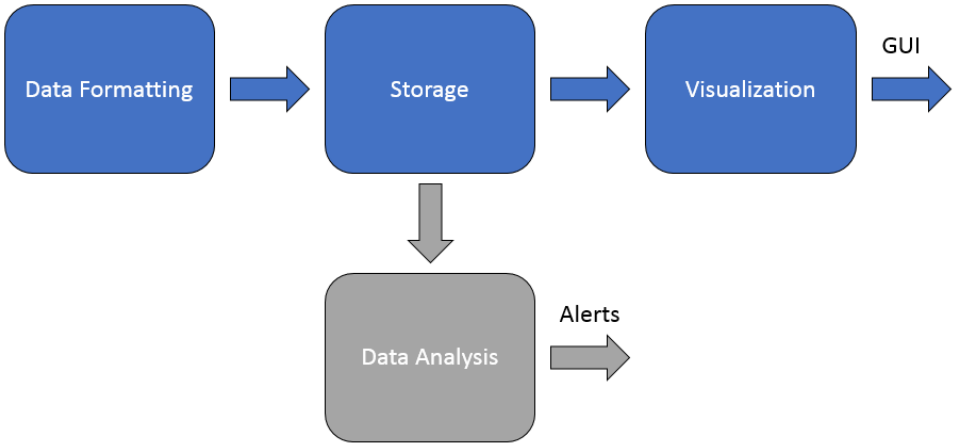

Figure 4.20 New Data Analysis component

By further developing the sample listener interface, new sample management and analysis techniques might be discovered. For example, improved sample querying could be used as a basis for the alerting and anomaly detection system. This approach is presented in figure 4.20 as a separate data analysis component. The benefit of creating such system as a separate component on top of the sample listener is that it would not be dependent on the implementation of the storage container.

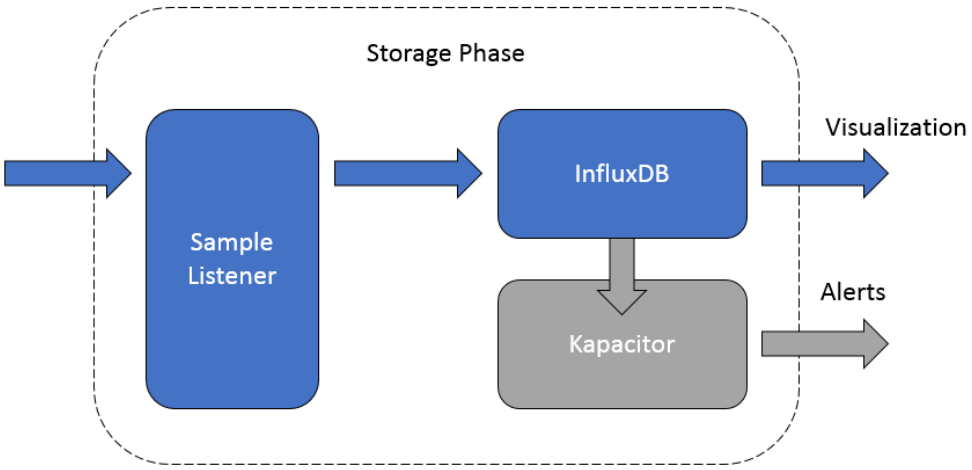

Figure 4.21 Data analysis using Kapacitor

Data analysis can also be done with less effort by utilizing the existing tools for InfluxDB. InfluxData provides the TICK stack for data analysis and visualization. Specifically, the Kapacitor tool can be used for anomaly detection and alerting as presented in figure 4.21. However, the usage of these tools means those parts of the system will be tied to InfluxDB and cannot be reused in case the storage implementation is changed.

As introduced in chapter 3.3.1, Grafana offers alerting functionalities as well. Currently, these functionalities are very basic and limited only to graph panels that do not use templating. As the development of Grafana continues, these functionalities may be improved. Since Grafana is compatible with numerous data storage implementations, the problem of getting stuck with a single solution is not as likely as with the TICK stack.

If the number of environments using the Data Visualizer system grows large enough, separate instances of the system might be required to visualize and combine the data to a sparser representation over longer periods of time. Both efficient application clustering techniques and data thinning methods must be studied to create a functional network for transferring data between the system instances.

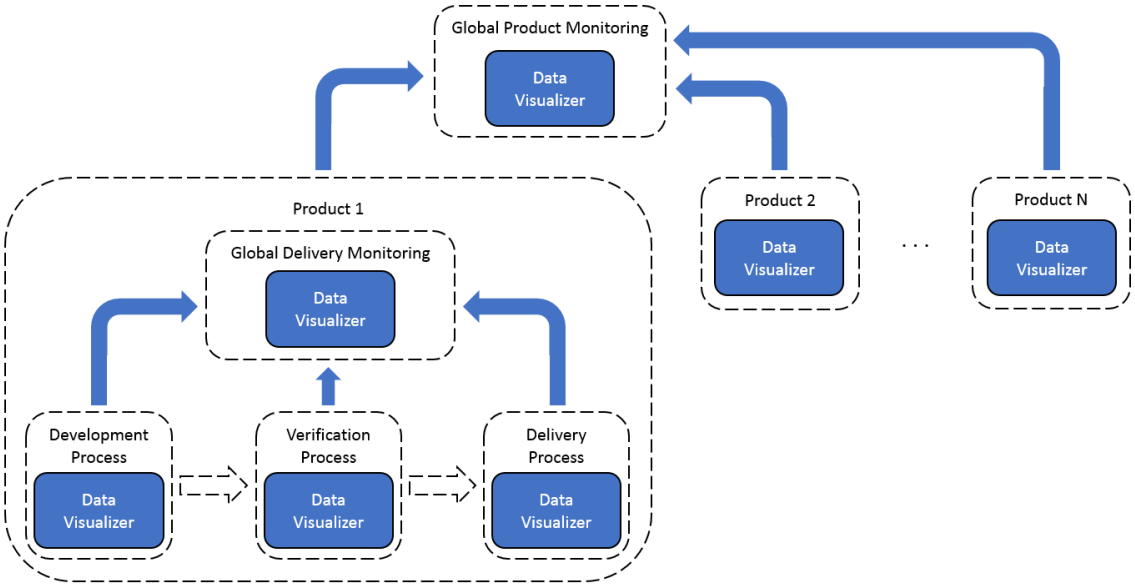

Figure 4.22 Scaling example for Data Visualizer

For example, the monitoring could be scaled on three levels as presented in figure 4.22. On the first level, different processes of the continuous delivery pipeline could be visualized in separate instances of Data Visualizer while the global pipeline status could be monitored in another instance that gathers thinned data from each lower level instance. The same structure could also be applied on a product level, where each separate product or separate versions of a specific product have their own delivery pipelines that are monitored on a global scale.

5. CONCLUSION

With the Data Visualizer system implemented and results from the two use cases analyzed, it is finally possible to compare the initial requirements with the gained results. The system is still under development, especially regarding the creation of shareable Grafana dashboards, and extensive user feedback has not been received yet. However, the potential benefits gained from the use of Data Visualizer are promising.

Since the implemented system covers only the required components for a data visualization system, there are numerous possibilities for expanding it with tools for further data analysis. Continuation of the system development while discovering additional ways to utilize the gathered data will be the focus of this project in future.

The initial requirement for this work was to implement a visualization service for a verification pipeline of a continuous delivery system. As the pipeline consisted of various other services and tools, it was important that the implemented system would be able to read and visualize data from all of them. The pipeline data was intended to be visualized as time series data over extended periods of time to help identify any notable trends in the status of the system.

The resulting Data Visualizer system fulfilled these requirements and was integrated with the pipeline environment without any major problems. Furthermore, the implementation was generic enough to be used to visualize any time series data from any environment, provided that a data format description and a data parser was implemented for the source of the data. Implementation of new data sources from the ground up took approximately one or two eight-hour workdays depending on their complexity.

Automatic configurations and the use of Docker environment helped to productize the system to a single package that was easy to set up and experiment with. It also greatly simplified creation of the additional tools like Grafana snapshot library and the sample backup scheduler.

Since Data Visualizer is a modular framework for different storage and visualization components, the obvious continuation for the development of the system is to add support for new components. For the storage component this would mean rewriting of the storage interface for each supported container. Respectively, to replace Grafana as the visualization component, rewriting of the visualizer interface would be required.

Although the data format description works well for the current implementation, new requirements might be discovered when the support for different storage containers are implemented. Users of the system might also give feedback on readability of the format syntax and suggest improved ways to represent structure of the written data.

As discussed in chapter 4.4.5, new components could be added to the system. Specifically, a separate data analysis component could be used for generating alerts or even automated feedback loops to the verification pipeline itself. By analyzing the Big Data analysis model presented in chapter 2.3, even more extensions to the system can be found.

The contents of this paper are targeted at anyone interested in utilizing the provided service. Therefore, it also functions an extensive documentation for the Data Visualizer service within the organization for those who are interested in knowing more than just the basic usage information provided in the user guides.

REFERENCES

[1] J. Humble and David Farley, “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, 1st edition, Pearson Education Inc. 2010.

[2] L. Chen “Continuous Delivery: Huge Benefits, but Challenges Too”, IEEE Software, vol. 32 no. 2 pp. 50 – 54, March – April 2015

[3] M. Meyer “Continuous Integration and Its Tools”, IEEE Software, vol. 31 no. 3 pp. 14 – 16, May – June 2014

[4] J. Humble et al. “The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations”, 1st edition, IT Revolution Press, Portland USA, 2016

[5] M. Hüttermann, “DevOps for Developers”, 1st edition, Apress, 2012

[6] S. Sharma “The DevOps Adoption Playbook: A Guide to Adopting DevOps in a Multi-Speed IT Enterprise” 1st edition, John Wiley & Sons Incorporated, 2017

[7] S. Paul “Continuous Delivery and DevOps: A Quickstart Guide”, 1st edition, Packt Publishing, 2012

[8] F. M. A. Erich, C. Amrit, M. Daneva “A qualitive study of DevOps usage in practice”, Journal of Software: Evolution and Process, Volume 29, Issue 6, June 2017

[9] L.E. Lwakatare, P. Kuvaja, M. Oivio, “Relationship of DevOps to Agile, Lean and Continuous Deployment”, Lecture Notes in Computer Science, Volume 10027, Springer, November 2016

[10] S. Diehl, “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software”, Springer, Berlin, Heidelberg, 2007

[11] W. Aigner et al. “Visualization of Time-Oriented Data”, Springer, London, 2011

[12] R. Mazza, “Introduction to Information Visualization”, Springer, London, 2009

[13] K. Hepworth “Big Data Visualization: Promises & Pitfalls”, Communication Design Quarterly, Volme 4 Issue 4, December 2016

[14] E. Zudilova-Seinstra et al. “Trends in Interactive Visualization”, Springer, London, 2009

[15] D. Gračanin et al. “Software visualization”, Innovations in Systems and Software Engineering, Volume 1, Issue 2, pp. 221-230, Springer, September 2005

[16] TM Forum Data Analytics Project, “Big Data Analytics Guidebook”, Release 16.5.1, June 2017

[17] S. Kandel et al. “Enterprise Data Analysis and Visualization: An Interview Study”, IEEE Transactions on Visualization and Computer Graphics, vol. 18 no. 12 pp. 2917-2926, December 2012

[18] M. Chen and A. Golan, “What May Visualization Processes Optimize?”, IEEE Transactions on Visualization and Computer Graphics, vol. 22 no.12 pp. 2619-2632, December 2016

[19] P. Caserta and O. Zendra, “Visualization of the Static Aspects of Software: A Survey”, IEEE Transactions on Visualization and Computer Graphics, vol. 17 no. 7 pp. 913-933, July 2011

[20] A.R. Teyseyre and M.R. Campo, “An Overview of 3D Software Visualization”, IEEE Transactions on Visualization and Computer Graphics, vol. 15 no. 1 pp. 87-105, January-February 2009

[21] M. Staron et al. “Measuring and Visualizing Code Stability -- A Case Study at Three Companies”, Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement, pp. 191-200, Ankara, Turkey, October 2013

[22] A. Leppäkoski and T.D. Hämäläinen, “PROMOTE: A Process Mining Tool for Embedded System Development”, Lecture Notes in Computer Science, Volume 10027, Springer, November 2016

[23] Seerene, “Applications”, Internet: www.seerene.com/applications

[24] “What is Power BI?”, Microsoft, Internet: powerbi.microsoft.com/en-us/

[25] Elasticsearch, “The Open Source Elastic Stack”,
Internet: www.elastic.co/products

[26] Steven Acreman, “Top 10 Time Series Databases”, 28.8.2016,
Internet: blog.outlyer.com/top10-open-source-time-series-databases

[27] Prometheus Authors, “Comparison to Alternatives”,
Internet: prometheus.io/docs/introduction/comparison

[28] InfluxData, “Time Series Database (TSDB) Explained”,
Internet: www.influxdata.com/time-series-database

[29] InfluxData, “InfluxDB 1.5 documentation”,
Internet: docs.influxdata.com/influxdb/v1.5

[30] Grafana Labs, “Grafana Documentation”, Internet: docs.grafana.org

[31] “Meet Jenkins”, 7.1.2016, Internet: wiki.jenkins.io/display/JENKINS/Meet+Jenkins

[32] “Robot Framework”, Internet: www.robotframework.org/#introduction

All internet sources were last accessed on 21.5.2018

APPENDIX A: DATA SOURCE DESCRIPTION SCHEMA

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1">
    <!-- datasource, sample and attribute name restriction -->
    <xs:simpleType name="nameType">
      <xs:restriction base="xs:string">
        <xs:pattern value="[a-z0-9_]+" />
      </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="customBoolean">
      <xs:restriction base="xs:string">
        <xs:enumeration value="true" />
        <xs:enumeration value="false" />
      </xs:restriction>
    </xs:simpleType>

    <!-- supported timestamp precisions -->
    <xs:simpleType name="precisionType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="ns" />
        <xs:enumeration value="u" />
        <xs:enumeration value="ms" />
        <xs:enumeration value="s" />
        <xs:enumeration value="m" />
        <xs:enumeration value="h" />
      </xs:restriction>
    </xs:simpleType>

    <!-- number of days for data expiration -->
    <xs:simpleType name="expireDays">
      <xs:restriction base="xs:integer">
        <xs:minExclusive value="0" />
      </xs:restriction>
    </xs:simpleType>

    <!-- attribute definition -->
    <xs:complexType name="attributeType" >
        <xs:attribute name="name" type="nameType" use="required" />
        <xs:attribute name="type" use="required" >
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="str"/>
                    <xs:enumeration value="int"/>
                    <xs:enumeration value="bool"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="min" type="xs:integer" />
        <xs:attribute name="max" type="xs:integer" />
        <xs:attribute name="allowed" type="xs:string" />
        <xs:attribute name="optional" type="customBoolean" />
    </xs:complexType>

...
```

```
...

    <!-- sample definition -->
    <xs:complexType name="sampleType">
        <xs:all>
            <xs:element name="identifiers" minOccurs="0" maxOccurs="1">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="attribute" minOccurs="0"
                            maxOccurs="unbounded" type="attributeType" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="measurements" minOccurs="1" maxOccurs="1">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="attribute" minOccurs="1"
                             type="attributeType" maxOccurs="unbounded" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="children" minOccurs="0" maxOccurs="1">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="sample" type="sampleType"
                                    minOccurs="0" maxOccurs="unbounded" />
                        <xs:element name="self" type="xs:string" fixed=""
                                    minOccurs="0" maxOccurs="1"/>
                    </xs:sequence>
                    <xs:attribute name="reference_identifiers"
                                  type="xs:string"use="required" />
                </xs:complexType>
            </xs:element>
        </xs:all>
        <xs:attribute name="name" type="nameType" use="required" />
        <xs:attribute name="timestamp_precision" type="precisionType" />
        <xs:attribute name="storage_instance" type="nameType" />
    </xs:complexType>

    <!-- format XML -->
    <xs:element name="datasource">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="sample" minOccurs="1"
                            maxOccurs="unbounded" type="sampleType" />
            </xs:sequence>
            <xs:attribute name="name" type="nameType" use="required" />
            <xs:attribute name="timestamp_precision" type="precisionType"
                          use="required" />
            <xs:attribute name="format_version" type="xs:integer"
                          use="required" />
            <xs:attribute name="expires_after" type="expireDays" />
        </xs:complexType>
    </xs:element>

</xs:schema>
```

APPENDIX B: DOCKERFILES

Sample Listener:

```
FROM python:3.5
WORKDIR /usr/src/app
COPY ./requirements.txt ./src ./startup.sh ./version ./
ENTRYPOINT ["./startup.sh"]
```

InfluxDB:

```
FROM influxdb:1.5.0
# User configurations
COPY ./influxdb.conf /etc/influxdb/influxdb.conf

# Forced Configurations
ENV INFLUXDB_META_DIR="/var/lib/influxdb/meta" \
    INFLUXDB_DATA_DIR="/var/lib/influxdb/data" \
    INFLUXDB_DATA_WAL_DIR="/var/lib/influxdb/wal" \
    INFLUXDB_HTTP_ENABLED="true" \
    INFLUXDB_HTTP_AUTH_ENABLED="true" \
    INFLUXDB_HTTP_HTTPS_ENABLED="false" \
    INFLUXDB_HTTP_MAX_BODY_SIZE="25000000"
```

Grafana:

```
FROM grafana/grafana:5.0.3
# Example dashboards
COPY ./dashboards/* /etc/grafana/dashboards/

# Provisioning configurations for example dashboards
COPY ./example_provider.yml \
     /etc/grafana/provisioning/dashboards/example_provider.yml

# User configurations
COPY ./grafana.ini /etc/grafana/grafana.ini

# Forced Configurations
ENV GF_PATHS_DATA="/var/lib/grafana" \
    GF_PATHS_PROVISIONING="/etc/grafana/provisioning" \
    GF_PATHS_LOGS="/var/lib/grafana" \
    GF_PATHS_PLUGINS="/var/lib/grafana/plugins" \
    GF_AUTH_BASIC_ENABLED="true"

# Required plugins
ENV GF_INSTALL_PLUGINS "grafana-piechart-panel 1.3.0,\
briangann-datatable-panel 0.0.6,\
natel-discrete-panel 0.0.7"
```

APPENDIX C: DOCKER COMPOSE FILES

docker-compose.yml:

```
storage:
  environment:
   - INFLUXDB_HTTP_BIND_ADDRESS=:8086
  build: ./storage/influxdb
  volumes:
   - ${DATA_DIR}/influxdb:/var/lib/influxdb:Z
  cpu_shares: 2048
visualizer:
  environment:
   - GF_SERVER_HTTP_PORT=${VISUALIZER_PORT}
   - GF_SECURITY_ADMIN_USER=${VISUALIZER_ADMIN}
   - GF_SECURITY_ADMIN_PASSWORD=${VISUALIZER_PASSWORD}
   - http_proxy=${http_proxy}
   - https_proxy=${https_proxy}
   - no_proxy=storage
  build: ./visualization/grafana
  ports:
   - ${VISUALIZER_PORT}:${VISUALIZER_PORT}
  volumes:
   - ${DATA_DIR}/grafana:/var/lib/grafana:Z
  links:
   - storage
listener:
  environment:
   - STORAGE_PORT=8086
   - VISUALIZER_PORT=${VISUALIZER_PORT}
   - VISUALIZER_ADMIN=${VISUALIZER_ADMIN}
   - VISUALIZER_PASSWORD=${VISUALIZER_PASSWORD}
   - LISTENER_PORT=${LISTENER_PORT}
   - http_proxy=${http_proxy}
   - https_proxy=${https_proxy}
   - no_proxy=storage,visualizer
  build: ./storage/listener
  ports:
   - ${LISTENER_PORT}:${LISTENER_PORT}
  volumes:
   - ${DATA_DIR}/listener:/var/lib/listener:Z
  links:
   - storage
   - visualizer
```

Environment configurations:

```
# System ports
LISTENER_PORT=****
VISUALIZER_PORT=****

# Grafana admin username / password
VISUALIZER_ADMIN=****
VISUALIZER_PASSWORD=****

# Directory where system data is stored
DATA_DIR=****

# Proxy settings can be defined here
#http_proxy=
#https_proxy=
```

APPENDIX D: JENKINS DATA FORMAT

```
<datasource name="jenkins" timestamp_precision="ms" format_version="1">
    <sample name="job">
        <identifiers>
            <attribute name="jenkins_name" type="str"/>
            <attribute name="name" type="str"/>
            <attribute name="result" type="str"
                       allowed="SUCCESS,FAILURE,ABORTED"/>
        </identifiers>
        <measurements>
            <attribute name="duration" type="int" min="0"/>
        </measurements>
        <children reference_identifiers="name,jenkins_name">
            <sample name="stage">
                <identifiers>
                    <attribute name="name" type="str"/>
                    <attribute name="result" type="str"
                               allowed="SUCCESS,FAILURE,ABORTED"/>
                </identifiers>
                <measurements>
                    <attribute name="duration" type="int" min="0"
                               optional="true"/>
                    <attribute name="running" type="bool"/>
                </measurements>
                <children reference_identifiers="name">
                    <sample name="step">
                        <identifiers>
                            <attribute name="name" type="str"/>
                            <attribute name="type" type="str"/>
                            <attribute name="result" type="str"
                                       allowed="SUCCESS,FAILURE,ABORTED"/>
                        </identifiers>
                        <measurements>
                            <attribute name="duration" type="int" min="0"
                                       optional="true"/>
                            <attribute name="running" type="bool"/>
                        </measurements>
                    </sample>
                    <self/>
                </children>
            </sample>
        </children>
    </sample>
</datasource>
```

APPENDIX E: ROBOT FRAMEWORK DATA FORMAT

```
<datasource name="robot" timestamp_precision="ms" format_version="1">
    <sample name="mainsuite">
        <identifiers>
            <attribute name="runner_name" type="str"/>
            <attribute name="name" type="str"/>
            <attribute name="passed" type="bool"/>
        </identifiers>
        <measurements>
            <attribute name="duration" type="int" min="0"/>
        </measurements>
        <children reference_identifiers="runner_name,name">
            <sample name="suite">
                <identifiers>
                    <attribute name="name" type="str"/>
                    <attribute name="passed" type="bool" optional="true"/>
                </identifiers>
                <measurements>
                    <attribute name="duration" type="int" min="0"
                               optional="true"/>
                    <attribute name="running" type="bool"/>
                </measurements>
                <children reference_identifiers="name">
                    <sample name="test">
                        <identifiers>
                            <attribute name="name" type="str"/>
                            <attribute name="passed" type="bool"
                                       optional="true"/>
                        </identifiers>
                        <measurements>
                            <attribute name="duration" type="int" min="0"
                                       optional="true"/>
                            <attribute name="running" type="bool"/>
                        </measurements>
                    </sample>
                    <self/>
                </children>
            </sample>
        </children>
    </sample>
</datasource>
```